TOWARDS A SELF-EVOLVING SOFTWARE DEFECT DETECTION PROCESS

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Xi Min Yang

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Software defect detection research typically focuses on individual inspection and testing techniques. However, to be effective in applying defect detection techniques, it is important to recognize when to use inspection techniques and when to use testing techniques. In addition, it is important to know when to deliver a product and use maintenance activities, such as trouble shooting and bug fixing, to address the remaining defects in the software.

To be more effective detecting software defects, not only should defect detection techniques be studied and compared, but the entire software defect detection process should be studied to give us a better idea of how it can be conducted, controlled, evaluated and improved.

This thesis presents a self-evolving software defect detection process (SEDD) that provides a systematic approach to software defect detection and guides us as to when inspection, testing or maintenance activities are best performed. The approach is self-evolving in that it is continuously improved by assessing the outcome of the defect detection techniques in comparison with historical data.

A software architecture and prototype implementation of the approach is also presented along with a case study that was conducted to validate the approach. Initial results of using the self-evolving defect detection approach are promising.

Acknowledgements

I am indebted to a number of people who have contributed to the successful completion of my Master's program. First, I would like to thank my supervisor, Dr. Kevin Schneider, for his guidance and valuable suggestions. In particular, I highly appreciate the great amount of time he spent for my program. I have learned a lot from his passion, dedication, and enthusiasm to research. My thanks also go to Dr. Chris Zhang (external examiner), Dr. Grant Cheston, and Dr. J.P. Tremblay for serving as members of my thesis defense committee. Their comments and suggestions help to improve the quality of the thesis.

Special thanks to Jan Thompson for her extra work which makes it easier for me to study at University of Saskatchewan while living and working in Regina.

Finally, I would like to express my profound gratitude to my wife, Florence Chen, my son, Max, my daughter, Melody, and my parents, for their love and incredible support.

To my beloved wife, Florence Chen.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Software defect detection is an important part of software development. The quality, the schedule, and the cost of a software product depend heavily on the software defect detection process. In the development of software systems, 40% or more of the project time is spent on defect detection activities [1, 4, 7, 20], such as, inspection, testing, and maintenance. In this dissertation, maintenance means the defect detection activities after software release, which include trouble shooting and bug fixing.

Software defect detection research has proposed new inspection and testing methods, and has studied and compared different inspection and testing methods. However, most of the research has focused on a single inspection or testing technique. At most, different inspection or testing techniques were compared to determine which one detected more defects. To be more efficient in this area, not only the study of a defect detection technique itself is necessary, but also more emphasis should be put on the defect detection process in which these techniques are applied. How can we get more from the defect detection process by a meaningful selection and combination of the available defect detection methods? How can we assess and improve the software defect detection process? To a large extent, these questions are still open. Since there is no general advice on how to conduct the software defect detection process, many medium and small software organizations apply some techniques based on personal preference and never use the other useful techniques at all. For example, testing may be used to the exclusion of inspection.

1.1 Motivation and Thesis Goals

This dissertation presents a self-evolving software defect detection (SEDD) model, and a prototype based on popular software process improvement models, such as the Capability Maturity Model [55], SPICE [19] and Bootstrap [31, 41, 66]. A process in these models is composed of a number of generic and base practices. The Generic practices would likely include: establish a defined process and measure the process results. The base practices of a testing process would likely include defect classification and defect detection.

SEDD helps improve the defect detection process by facilitating defect detection technique selection and assessing defect detection results. Defect detection technique selection is facilitated by defining the base practices of the defect detection process, and defining each base practice's entry criteria (the criteria that must be met before a practice can be applied) and exit criteria (the criteria that must be met before a practice is considered complete). SEDD helps assess the defect detection process by providing an in-process feedback mechanism using historical data and data collected during the defect detection process. The model also provides improvement functionality by referencing the experience base to provide information on which action should be taken.

Another important aspect of the SEDD is its self-evolving feature. This means the model starts simple; at first, its defect classification scheme, its defect database, and its experience base should be as simple as possible, and the base practices should be as few as possible so that SEDD is easy to apply. The in-process evaluation function continuously improves the model in three ways: (1) by adding important elements and dropping the unimportant elements from the defect classification scheme; (2) by adjusting its entry criteria and exit criteria for the base practices; and, (3) by enriching and consolidating its experience base to better fit the requirement of an organization, department, or a project.

This model can help the software development team to answer the following questions: Which defect detection technique should be used at a specific stage? How

well does the defect detection process work? What is the strength of the current software defect detection process? Where is the weakness of the current software defect detection process? What needs to be done to improve the defect detection process?

The motivation of this thesis is to present a self-evolving software defect detection process model that provides a mechanism to combine defect detection techniques, to assess the previous and current defect detection practice, and to adjust and improve the defect detection process. In particular, the goals of this research are:

- To propose a systematic approach to the software defect detection process.
- To present a self-evolving software defect detection process model based on the proposed systematic approach.
- To build a prototype for a self-evolving software defect detection control system based on the model.
- To perform a case study to evaluate the new approach.

1.2 Thesis Overview

This research presents a model for a self-evolving software defect detection process that uses inspection, testing, and maintenance in combination, instead of in isolation, to achieve better results. This approach enables us to conduct and control the defect detection process better by establishing and refining entrance criteria and exit criteria. This approach also can help us evaluate the software defect detection process by analyzing the defects from the process and their comparison with historical data. This approach provides a self-evolving mechanism as well, by identifying the weak points in the current process and improving it accordingly.

The first part of the thesis describes the new approach. This new approach aims to help us get more from the defect detection process through a meaningful selection and combination of the available defect detection methods. In this part, different

software defect detection techniques are studied and classified. Special attention is paid to the fact that each technique is good at detecting some specific types of defects. It is importing not to decide which technique can detect more defects than the others, but how can these techniques be put together and how can they complement each other so that a better overall cost-effectiveness can be achieved.

To achieve the goals set by the new approach, a model is presented for the software defect detection process. This model contains the necessary components and mechanisms that enable us to decide when to apply inspection and when to apply testing by defining the entry criteria and exit criteria for each base practice. It also helps us to assess the previous and current software defect detection activity through the in-process feedback mechanism using the defect data collected during the process, and helps us to adjust and improve the process using the historical data in the experience database.

Based on the model, a prototype for the self-evolving defect detection process control system is built to demonstrate the feasibility of the new approach in the software industry. The structure and main functions of the system are discussed, the design and implementation considerations are addressed, the technologies used are covered, and the results are discussed.

The remainder of this thesis is organized as follows. Chapter 2 introduces the background for the study and reviews the related work. Chapter 3 describes the new approach to the software defect detection process. Chapter 4 presents the model built to demonstrate the applicability of the new approach: section 4.1 defines the major components of the framework, and section 4.2 defines the functionalities of the components and the relationship between these components. In Chapter 5, a prototype was built for the new approach. A case study was conducted and is described in Chapter 6 to evaluate the new approach. Chapter 7 summarizes the dissertation and proposes future work.

Chapter 2

BACKGROUND AND RELATED WORK

Because of the vital role it plays in software product development, software defect detection has stimulated research interest for decades. New software defect detection methodologies have been introduced to provide more effective and less costly techniques. This chapter first provides some background into software defects, software defect detection techniques, defect detection classification schemes, software process improvement models, the testing maturity model, and defect detection classification. Then it reviews previous studies on two of the most common software defect detection activities: inspection and testing.

2.1 Definitions and Terminology

2.1.1 Software Product

A software product is any artifact created as part of creating and maintaining software, including computer programs, plans, procedures, and associated documentation and data [55].

2.1.2 Software Process

A software process is a set of activities, methods, practices, and transformations that people use to develop and maintain software products [38].

2.1.3 Software Defects

A software defect is any flaw or imperfection in a software product, including both code and documentation.

2.1.4 Maintenance

Software maintenance is the activity required to keep the software system functioning properly or to add enhancements after software release.

2.1.5 Software Defect Detection

Software defect detection is the process of discovering software defects, commonly, it includes these activities: inspection, testing, and maintenance.

2.1.6 Software Defect Detection Effectiveness

Software defect detection effectiveness is the percentage or ratio of the number of the defects detected to the number of all the defects contained in a software product.

2.1.7 Software Defect Detection Efficiency

Software defect detection efficiency refers to the number of defects detected in a time unit (per day or per hour).

2.1.8 Entry Criteria

Entry criteria are the predefined requirements that must be met for an activity to start [21]. For example, for a testing process to commence, the tester must be available, the code to be tested must be implemented, the test cases must be built, and the test environment must be set up.

2.1.9 Exit Criteria

Exit criteria are the predefined conditions used to verify that an activity completes. For example, when testing software, a test requirement is created to verify the software meets operational requirements. This test requirement sets the conditions necessary for the testing process to be considered complete. For example, one of the exit criteria for testing could be that the defect removal ratio should be higher than 90%.

2.1.10 Base Practice

A base practice is a key practice in either a software engineering or management activity. For example, the base practices of the testing process include entry criteria checking, defect detection technique selecting, executing, defect gathering, defect classification, defect analyzing, and exit criteria checking.

2.2 Defect Classification Schemes

Since 1975, a number of classification schemes have been developed by different organizations, such as HP and IBM, to classify software defects and to identify common causes for defects in order to determine corrective action.

2.2.1 Hewlett-Packard - "Company-Wide Software Metrics"

Hewlett-Packard[29, 30] classifies defects from three perspectives in three steps (cf. Figure 2.1): (1) identifying where the defect occurred (e.g., in the design or the code); (2) finding out what was wrong (e.g., the data definition or the logic description may be incorrect); and, (3) specifying why it was wrong, missing or incorrect.

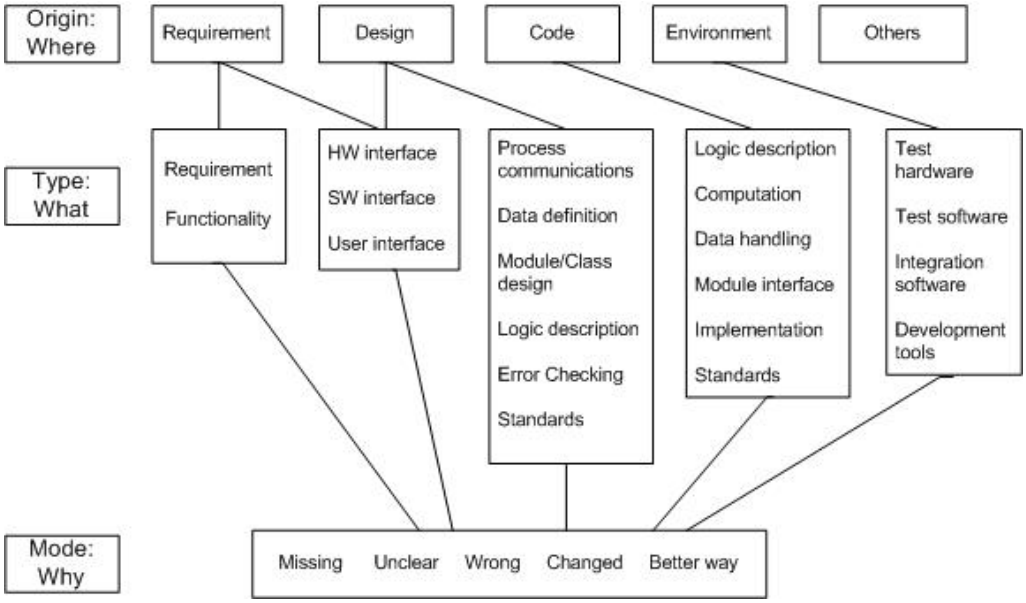

Figure 2.1: HP Classification Scheme (adapted from [29])

2.2.2 The IBM Orthogonal Defect Classification Scheme

The IBM Orthogonal Defect Classification (ODC) was originally described in the paper by Chillarege et al. in 1992 [12]. As described by Chillarege, the goal of ODC is to provide a scheme to capture the key attributes of defects so that mathematical analysis is possible. The software development process is evaluated based on the data analysis. According to ODC, the defect attributes that need to be captured include: defect trigger, defect type, and defect qualifier. The "defect type" attribute describes the actual correction that was made. For example, if the fix to a defect involves interactions between two classes or methods, it is an interface defect. The "defect trigger" attribute represents the condition that leads the defect to surface. For example, if the tester found the defect by executing two units of code in sequence, the defect trigger is "Test sequencing". "The defect qualifier" indicates whether the defect is caused by a missing or wrong element.

2.3 Software Inspection

2.3.1 Definition

There is more than one definition for Software Inspection; some of the well accepted definitions are:

- "An inspection is a static analysis technique that relies on visual examination of work products to detect defects, violations of development standards, and other problems" [38].
- "An inspection is a formal review of a work product by the work product owner and a team of peers looking for errors, omissions, inconsistencies, and areas of confusion in the work product" [65].
- Fagan Inspection [22] refers to inspection as a structured process of finding defects in specifications, design documents, and code during the software development process.

From the above definitions, we can see that inspection refers to a structured peer review of a software product to look for defects using a well defined process. Inspections can be used at every level of the software development process to review requirements, designs, code, and even test cases.

2.3.2 Roles in software inspection

In realizing that software architects/developers are often blind to the defects in their own work and the inefficiency of informal review, Fagan [22] proposed the use of formal inspections which are conducted in a rigorous process by a group of people each with a specific role. The different roles within the inspection process [22]:

- Designer: the author of the design document.
- Coder: the programmer who implemented the design with code.

- Tester: the person who wrote the test case or the person who did the testing.
- Moderator: the person who leads and manages the inspection.
- Meeting logger: the person who documents the meeting minutes.

2.3.3 Inspection process

The inspection process consists of the following operations, which are all needed for effective inspections [22, 18]:

Planning

- Preparation of materials to be inspected
- Selection of inspection participants
- Scheduling of inspection meeting (include the time and the place)

Overview

- The introduction of the product to be inspected to the participants by the designer.
- Assignment of roles

Preparation

- The work that the participants do to help them get familiar with the product to be inspected and prepare themselves for their roles

Inspection

- The participants read through the document/code to uncover the defects

Rework

- The work performed by the author to resolve the defects found by the participants during the inspection phase.

Follow-up

- The phase in which the moderator verifies that all defects found in the inspection phase are fixed and no new defects are inserted in the rework phase. It is important that all defects are corrected as early as possible, as the costs of fixing them in a later phase of the project have been shown to be 10 to 100 times higher.

2.3.4 Inspection Techniques

Since Fagan introduced inspection into the software development process, many inspection techniques have been developed. Among them are the following:

Fagan's Software Inspection

To improve the defect detection and removal efficiency, Fagan [22] defines software inspection to be a rigorous review. Fagan's inspection has a predefined process which includes: overview, preparation, inspection, rework, and follow-up. Also, with Fagan's inspection, each participant has a predefined role: designer, coder, tester, and moderator, and meeting logger. As well, Fagan's inspection is meeting-oriented.

Peer Reviews

Typically, people are blind to their own mistakes and do not like their mistakes to be known by a person at a higher level in their organization. Based on this observation, peer review was introduced. In a peer review, the software artifact is reviewed by one or more colleagues at the same level in the organization as the designer/developer.

Formal Reviews

Considering that the ordinary inspection techniques may not be rigorous enough to be effective, formal reviews were proposed by Weinberg and Freeman [25]. In a formal review, the review standards are given to the reviewer and the reviewer should have a clear idea on what to review and how to review the software artifact. After review, a structured report is developed describing the result of the review.

FTArm

To overcome the shortcomings of the formal technical review, such as: significant expense, clerical overhead, group process obstacles, and research methodology problems, Johnson [35] proposed FTArm (Formal Technical Asynchronous review method) to improve the effectiveness and efficiency of conventional formal technical reviews with parallel activity and computer support. This technique is composed of six phases: setup, orientation, private review, public review, consolidation, and group review.

N-fold Inspection

N-fold inspection was proposed by Schneider et al. [63] based on the hypothesis that N separate inspection teams do not significantly duplicate each others' efforts, so that the total number of faults detected will be much higher than the number found by any one team during a single inspection. In their controlled experiment, they carried out nine formal inspections of a document in parallel and the result confirmed earlier conclusions that the N-fold inspection method is more effective than a single inspection.

Two-person Inspection

Bisant and Lyle [6] investigated the effect of a two-person inspection method on programmer productivity. The two-person inspection advantage is that it is less costly than the conventional Fagan's inspection since it only involves two people, the author and one reviewer. This two-person method could have its application in those environments where access to larger team resources is not available.

Phased Inspection

Considering that most of the inspection techniques are not formal enough to be dependable, Knight[40] proposed a new review method, Phased Inspection. With phased inspection, the inspection process is divided into several parts (phases) and

each phase focuses on a specific aspect. Phased inspection is developed to bring more formality, reliability, and repeatability into an inspection process.

2.4 Testing

Software testing is a process to verify the correctness, completeness, and quality of a software product. Testing verifies a software product dynamically by executing the product in its working environment, while inspection checks it statically.

2.4.1 Testing strategies

To make testing both more effective and efficient, it has to be conducted in a strategic way. The most common used approaches are bottom-up and top-down.

Bottom-up. Bottom-up starts with the smallest components, units, which might be a method or a class. Each of the units is tested individually. After unit testing, module testing is performed. A module is a collection of units, for instance a class. After module testing is sub-system testing, in which sets of modules are integrated into a sub-system and tested together. Interfaces defects are often discovered when sub-system testing is performed. Sub-systems are integrated together to validate that all the whole system performs correctly.

Top-down. The top-down approach is the opposite of the bottom-up method. With this approach, the top level modules are developed and tested first, and then testing continues with the lower levels. Top-down testing can be performed in two ways: breadth first or depth first. Depth first means starting at the top level and then following a path all the way down to the bottom, one level at a time. Breadth first means starting at the top level and after developing and testing all the units at this level and then moving down to the next level.

2.4.2 Testing techniques

As a primary and basic software verification and validation approach, Software testing has always attracted numerous researchers since Turing [70] published his paper on "checking a large routine" in 1950. Although numerous types of testing have been defined, they mainly can be grouped into two categories: black-box testing and white-box testing.

Black-box testing

Black-box testing focuses on the functional requirements of the software product from a user's perspective. So it is most used in functional testing phase. Black-box testing enables the tester to find missing or incorrect functions, interface errors, performance issues, setup and exit problems without knowing the internal detail of the software product.

There are three common black-box methods often used for function-based tests [39]: equivalence partitioning, boundary value analysis and error guessing.

- **Equivalence partitioning**: Equivalence partitioning divides the input (sometimes even output, although it is rare) into sections. One value from each section is chosen and used as a representative of the whole section for testing. As described in [54], equivalence class partitioning can be quite a systematic approach to black-box testing.

- **Boundary values analysis**: Boundary values analysis focuses on checking the lower and upper limits of each section [54]. Since the boundaries of input sections are the areas where developers are prone to making errors, boundary values analysis helps detect any defect at these boundaries.

- **Error guessing**: Error guessing is a technique used by an experienced tester to generate test cases based on the tester's intuition and experience. With their knowledge of the error-prone areas and the types of faults to expect, testers design test cases specifically to expose defects.

White-box testing

The goal of white-box testing of source code is to identify infinite loops, missing paths [23], and dead code.

White-box testing requires knowledge of the internal details of the code to be tested, its structure and its logic. It is most often used in unit testing. With white-box testing, the software engineer can verify the code by coverage, path, or condition testing. Four basic forms of logic coverage when applying white-box testing are [45]:

- **Statement coverage**. Statement coverage requires that each statement be executed at least once.
- **Decision coverage (branch)**. Decision coverage [54] requires each decision (e.g. If statement, While loop statement) be evaluated with both "True" and "False" at least once.
- **Condition coverage**. Condition coverage [54] requires that all atomic boolean conditions in combined expressions to be evaluated with both "True" and "False" values at least once
- **Path coverage**. Path coverage [5], [13], [47]-[49] requires that each possible combination of branches from the entry of a method to the exit be executed at least once.

2.4.3 Types of testing

The types of testing include the following:

Unit testing: Unit testing is the lowest level test. It is a procedure to verify that a single component of source code is working properly.

Integration testing: Integration testing combines two or more units as groups to test the interfaces between them and to verify that they are compatible with each other. While unit testing focuses on the behavior of a single unit, integration testing assure the communication between different units works properly.

System testing: System testing is the highest level of testing. It tests the functionalities of the entire system and verifies a system's compliance with its specified requirements. System testing focuses on synchronization and timing errors, volume/load/stress problems, shared resource conflicts, and security problems.

Acceptance testing: Acceptance testing is the test performed by the end users to validate whether the system satisfies their requirements. Based on the testing results, the users decide whether or not they will accept the system.

2.5 Comparison of Different Defect Detection Techniques

In the past three decades, there have been numerous publications on software defect detection techniques, and most of them address a single defect detection technique. In recent years, there has been research comparing different defect detection techniques. However, the number of these kind of studies is quite small, and the focus of the research is usually situated in the comparison of closely related techniques: such as Porter and Votta's [59] study on comparison of different reading techniques; Macdonald, F. and J. Miller's [44] study on "tool-based and paper-based software inspection."; and Hetzel[32] and Myers's[53] study on comparison of: black-box testing, white-box testing, and individual code reading. This section reviews previous studies that compare different defect detection techniques.

2.5.1 Comparison of Different Inspection Techniques

The three most commonly used inspection techniques are: ad hoc reading, checklist-based reading, and scenario-based reading. Ad hoc reading provides no instructions on how to read, and is fully dependent on an inspector's personal preference and experience [57]. Checklist-based reading provides inspectors with a list of questions to be answered [22, 21, 1, 33, 68, 28]. Similar to ad hoc reading, checklist-based reading leaves inspectors to decided how to check the items on the list. Although

it supports inspectors better than ad hoc reading, checklist-based reading has three serious weaknesses. First, the questions on a list are from previous projects, the literature, or other organizations, and quite often they are too general and not sufficiently tailored to the document under review. Second, every inspector uses the same checklist and is expected to answer all the questions on the list. Quite often a person who is an expert in one area may not have much knowledge in another area, so it is unrealistic to expect an inspector to give proper answers to all the questions which cover different areas of the system. Third, inspectors concentrate on the types of defects on the list, without considering the types of defects not listed on the list.

More recently, scenario-based reading [57, 2] was introduced to address the above shortcomings. Scenario-based reading uses scenarios to describe how to read and what to look for. A scenario denotes a procedure that a inspector should follow. For example, for an e-business company, a scenario built from the end users' perspective could be: check if the response time is acceptable, check if the web pages are attractive, check if it is easy to find a specific item, and check if it is convenient to pay. So far, two different types of scenario-based reading have been proposed: perspective-based reading and defect-based reading. Perspective-based reading proposes that a software product should be inspected from the perspective of different stakeholders [3, 42]. Defect-based Reading is for different inspectors to focus on different defect classes (such as backward compatibility or document consistency) while reading a requirements document [60, 51]. With this approach, a scenario is created for detecting a particular type of defect, which directs inspectors to detect corresponding types of defects.

To evaluate which of these techniques is more effective, various experiments have been performed. The result of this research has not reached a consensus. Some experiments showed that scenario-based reading (SBR) is more effective than other reading techniques. For example, Basili et al. performed an experiment with professional software developers from the National Aeronautics and Space Administration / Goddard Space Flight Center (NASA/GSFC) [3]. The results showed that the scenario-based reading technique, perspective-based reading, was more effec-

tive than other reading techniques. Their conclusions were supported by several experiments [58, 14, 56, 4, 43, 69]. However, the results from other experiments [16, 17, 26, 51, 62, 61] contradict Basili's findings. Some experiments found that checklist-based reading is more effective, some experiments found that there is no significant difference between the reading techniques, and some experiments even found ad hoc reading is more efficient.

The rest of this section reviews the research comparing defect detection inspection techniques.

Porter, Votta, and Basili

Considering that two of the most commonly used inspection techniques, ad hoc reading and checklist-based reading, are not systematic, and the reviewers have to read a whole document to uncover all types of defects with no guideline on how to proceed, Porter, Votta, and Basili [60] theorized that scenario-based reading will perform better results if each reviewer used specific techniques to search for specific types of defects. They evaluated this hypothesis with forty eight graduate students in computer science. These student were grouped into sixteen teams and each team used some combination of ad hoc, checklist or scenario methods. The experimental results are:

- Overall, the scenario-based reading uncovered more defects than either ad hoc reading or checklist-based reading.
- Scenario reviewers were especially effective at detecting the type(s) of defects for which their scenarios were created, and were not as effective as ad hoc or checklist-based at detecting other types of defects.
- There was no difference between checklist-based reading and ad hoc reading with respect to effectiveness.
- Contrary to Fagan's finding, collection meetings had no effect on the defect detection rate.

Basili et al.

To evaluate the effectiveness of Perspective Based Reading (PBR), Basili et al. [3] conducted a controlled experiment with professional software developers from the National Aeronautics and Space Administration / Goddard Space Flight Center (NASA/GSFC) Software Engineering Laboratory (SEL). The subjects read two types of documents, one generic in nature and the other from the NASA domain, using two reading techniques, a PBR technique and their usual technique. The results from these experiments are:

- With respect to the overall performance of a team, perspective-based reading had significantly better coverage of both generic and NASA specific documents.
- With respect to the performance of individuals, perspective-based reading uncovered about the same amount of defects as the others techniques in NASA specific documents, but performed significantly better on generic documents.

From the above results, we can see that the performance of PBR and ad hoc varied with types of documents.

Ciolkowski, Differding, Laitenberger, and Munch

Ciolkowski et al. [14] conducted a replication of Basili's experiment within an academic environment to validate the original results from NASA/Goddard Space Flight Center. The results are:

- Perspective-Based Reading was more effective than ad hoc reading for both individual and team.
- There is no significant difference between the team of programmers and the team of students.

Fusaro, Lanubile, and Visaggio.

Fusaro et al. [26] replicated the experiments of Porter, Votta, and Basili with two runs of a controlled experiment with over one hundred undergraduate students taking

a software inspection course. In their experiments, they compared perspective-based reading with ad hoc reading and checklist-based Reading in the context of both individual and team. They also analyzed the effects of combining different or identical perspectives. The reviewers used the same requirements documents and followed the same procedure as the previous studies. The students reviewed the documents by either applying an unsystematic reading technique (ad hoc in the first run and checklist in the second run) or a systematic reading technique (PBR). The findings from their experiment are the following:

- The inspection teams applying PBR found the same ratio of defects as the inspection teams applying ad hoc or checklist-reading.
- Individuals using PBR technique found a smaller percentage of defects than individuals using ad hoc reading technique.
- There was no difference between individuals using the PBR technique and individuals using the checklist technique.
- The teams where each member has an identical role uncovered the same percentage of defects as the teams where each member has a different role.

Sandahl, Blomkvist, Karlsson, Krysander, Lindvall, and Ohlsson.

Sandahl et al [62] conducted an experiment that was a replication of the Porter, Votta, and Basili experiment comparing the defect-based reading scenario method and the checklist-based reading method in the same context. The reviewers were undergraduate students and the document under review was a requirements specification. The result of their replication experiment was contrary to the original experiment:

- DBR reviewers did not have significantly higher defect detection rates than checklist reviewers.

This finding is in accordance with a replicated experiment conducted by Fusaro, Lanubile and Visaggio [26], but is contradictory to the original experiment [60].

Miller.

Miller et al. [51] replicated the experiment performed by Porter [59] comparing Defect Based Reading technique and the checklist-based technique. Their experiment used the same documents as Porter's, WLMS and CRUISE and the results are:

- DBR was not more effective than checklist in the WLMS document.
- DBR was not more effective than checklist in the CRUISE document.

The above results are ambiguous, but on balance are generally supportive of the results in the original experiment.

Halling, Biffl, Grechenig.

Halling et al. [50] performed a large-scale experiment in an academic environment. The experiment evaluated the effectiveness of defect detection for inspectors who use a checklist or scenarios at both an individual and a team level. Two of their findings are:

- The checklist significantly outperformed the scenarios on an individual level.
- The scenarios were more effective regarding their target focus.

Halling's findings show that checklist and scenarios both have different strong areas.

2.5.2 Comparison of Different Testing Techniques

The research on the comparison of testing technique traces back to as early as three decades ago. In 1976, Hazel [32] designed and conducted a controlled experiment in order to analyze three basic verification methods: reading, specification testing (functional testing), and selective testing (a variation of structural testing). In Hazel's experiment, 39 subjects verified three structured PL/I programs in sessions. In each session, the subjects checked one of the programs using one of the three techniques. His main findings were:

- The subject did not find as many defects using reading as using the other two techniques.
- Functional testing and structural testing were equally effective.
- On the average, only little more than half of the errors were found.
- It is not possible to find all the defects in a product by using only one technique.

This work was replicated by Myers [53]; he performed a study on the comparison of the three defect-detection techniques: reading, functional testing, and structural testing with respect to their effectiveness and efficiency at detecting defects. The experiment employed 59 experienced professional programmers to test a small PL/I program. The results show:

- The techniques were not different in the number of defects they detected; reading was as effective as the other two computer-based methods in finding errors.
- Code reading was less cost-effective than the others.
- All pairing of techniques were superior to single techniques.
- The number of defects found varied dramatically from person to person.
- The types of defects found varied dramatically from method to method.

With the motivation to improve and better understand defect detection techniques, Basili and Selby [4] conducted a study to characterize and evaluate these three techniques. The 74 subjects (some are students and the others are professional programmers) tested four unit-size programs and the results of the experiments were:

- Code reading uncovered more defects than other techniques.
- The performance of these three techniques varied with software type.

Selby [64] compared the six pair-wise combinations of three common testing strategies – code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing with 100%

statement coverage criteria – among themselves and versus the individual techniques. The major results of the study are the following:

- The combined testing strategies uncovered significantly more defects than did the single techniques.

- The combinations of two code readers or one code reader and one functional tester uncovered the highest rate of defects.

- The expertise level of a tester gave a statistically significant result; senior testers detected more defects than junior testers.

- Both the effectiveness and the efficiency of a test technique depended on the type of software under being tested.

Kamsties and Lott [36] extended the design and techniques originally used by Basili and Selby and replicated the Basili and Selby experiment twice. They found:

- These three testing techniques were similarly effective in detecting defects.

- The code reading and the functional testing isolated approximately the same percentage of faults, with the structural testing performing less well.

- Functional testing identified the existence of defects quicker than code reading, but required more time to locate the defect than code reading.

- Overall functional testing was more efficient than code reading.

The results from previous research show that there is not a specific inspection or test technique that is more effective than the others in general. Instead, a technique that performs better in one experiment may be inferior in another experiment. So the findings from previous research demonstrate the necessity of combining the different techniques to achieve better overall results.

2.6 Limitations of Current Research on Software Defect Detection

From the above review of controlled experiments, one may draw the conclusion that there is neither a consensus conclusion on the comparison of different inspection techniques, nor a consensus result on the different testing techniques. In other words, there is no clear, consistent evidence that one defect-detection technique is better (more effective or more efficient) than the others when used independently. The different, even contradictory results, from the experiments could be caused by the limitations of the previous research on software defect detection:

- Their experiments were conducted in different contexts: different documents/programs, different types of defect detection, different checklists/scenarios, different inspectors/testers, different experiences and different familiarity with the documents/programs. So the conclusions they drew may only apply to their specific experimental environment and are not general enough to be applied to other research environments, let alone to the information technology industry.

- The research focused on the comparison of similar techniques, such as different inspection techniques: ad hoc reading, checklist reading and perspective-based reading; or different testing techniques: functional testing and structural testing. They treated the different techniques as rivals, rather than complementary to each other.

- The research concentrated on a specific phase of the defect-detection process. They treated inspection and testing in isolation, rather than by treating the defect detection process (including inspection, testing, and maintenance) as a whole. Their research was centered only on evaluating which reading technique was more effective at the inspection stage, without evaluating which one is more effective when combined with testing. Similarly, they only evaluated which testing technique detected more defects than others at the testing stage,

failing to consider which testing technique is better than others when it is applied with inspection.

- Most of the previous work only considered the effectiveness of different defect-detection techniques when doing a comparison, without taking efficiency into account.

- Previous work used the number of defects that a defect-detection technique detected as the gauge to decide which technique is better than others, without incorporating the severity of the defects and the cost to fix the defect, if it is not detected.

2.7 A Different Approach from Current Research

In an effort to overcome the above weaknesses, this thesis takes a different approach to the study of software defect detection. Compared with the current research, this approach has the following aspects:

- Instead of trying to draw a conclusion on which defect-detection technique is better than others based on an experiment conducted in a specific environment, this thesis presents a model by which both the different defect detection techniques and the entire defect-detection process can be evaluated.

- The model is intended to be general enough to be applied to different organizations.

- The model can be tailored to suit the different requirements of an organization.

- The model evaluates the different defect-detection techniques in the context of the entire defect-detection process, including inspection, testing, and maintenance.

- The model evaluates the different defect-detection techniques not only based on the defect-detection effectiveness, but also the efficiency, and the risk of failing to detect the defect.

The next chapter will present a model expanding on these aspects.

Chapter 3

THE SELF-EVOLVING DEFECT DETECTION PROCESS(SEDD)

This chapter describes a new approach to software defect detection to better conduct, control, evaluate, and improve the defect detection process. The necessity of a new approach and the rationale behind the approach are discussed in Section 3.1 through Section 3.3. After discussing the controlling of the software defect detection process in the second part of Section 3.3 and defect classification schemes in Section 3.4, the evaluation mechanism of the new approach is introduced in Section 3.5 and Section 3.6. Finally, the improvement mechanism of the new approach is presented in Section 3.7 and Section 3.8.

3.1 The Systematic Approach to Software Defect Detection

The software development process is made up of a number of activities: analysis, design, development, testing, and maintenance. Although the main activities in these phases are different from each other, there are internal relationships among them. These activities should not be treated in isolation and should not be limited to a specific phase. This is especially true with software defect detection. It is not and should not be limited to testing, and instead, it is an activity that should be carried out through the entire software system life cycle. Although in different phases the activities of software defect detection are given different names: inspection, testing, or maintenance; intrinsically they are the same thing under a different name. So to

improve software defect detection, all of the three activities: inspection, testing, and maintenance, should be addressed as a whole.

3.2 The Necessity of a New Approach

From the review of the related work in the last chapter, we can see that most of the current research treats inspection and testing in isolation, with some solely focusing on inspection and others solely focusing on testing. Although some research relates to both inspection and testing, they treat these two as rivals, rather than complements of each other. They compared different types of inspection and testing, and tried to draw a conclusion on which one is more effective than the other. As well, the third essential defect detection technique, maintenance, was ignored by the research. As a result, the research fails to realize the relationship between inspection, testing, and maintenance, and fails to give guidance on controlling, evaluating, or improving the defect detection process.

3.3 The Rationale of Applying More Than One Defect Detection Technique in Combination

From the review of the related work in Chapter 2, the following conclusions can be drawn:

- It is impossible to detect all the defects in a product by applying only one kind of defect detection technique.
- It is impractical to apply inspection to the extent that all the defects which could be detected by inspection are detected before moving to the testing stage.

So the art of controlling the defect detection process is to decide which kinds of defects should be detected by inspection, which types of defects should be detected by testing, which types of defects should be (or have to be) left to maintenance, when to apply inspection, and when to apply testing.

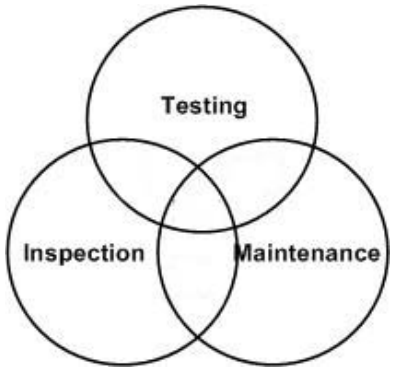

Figure 3.1: Defect Detection Techniques' Coverage of Defect Types

To answer these questions, a more systematic approach should be adapted. Instead of considering only part of the process separately, we should address the software defect detection process as a whole by combining inspection, testing, and maintenance.

As we know, of all the types of defects contained in a software product, some can only be detected by inspection (e.g., design conformance defects and algorithm defects); some can only be detected by testing, (e.g., timing and serialization defects), some can only (or have to) be detected by maintenance after release (e.g., some of the rare/ abnormal situation handling defects), while other defects can be detected by either inspection, testing, or maintenance (cf. Figure 3.1). For those types of defects that can be detected by more than one defect detection technique, we need to find out which technique is more efficient. Therefore, in software defect detection, it is not as important to find out which technique is more effective as it is to find out the best way to combine inspection, testing, and maintenance to conduct the defect detection process most effectively and most efficiently. In other words, we want to achieve a better coverage with less cost.

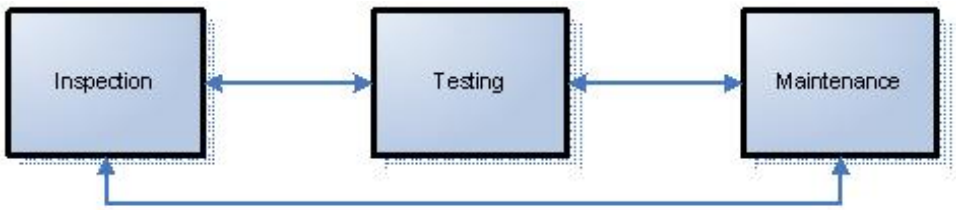

Figure 3.2: The Integrated Approach to Defect Detection

To achieve the goal of the best coverage with least cost, an integrated approach (Figure 3.2) should be used so that each technique is used in the area in which the technique performs better than the others. To further benefit from the integrated approach, a self-evolving mechanism can be incorporated to evaluate and improve the techniques. On one hand, by analyzing the defect data collected during the inspection, it is possible to provide a guideline to testing. On the other hand, by analyzing the defect data (the defects which are detected by testing but should have been detected by inspection), it is possible to evaluate inspection and give feedback on improving inspection.

Including maintenance as part of the software defect detection process allows for the defect data obtained during maintenance to be used to evaluate and improve inspection and testing. If the defect type distribution for maintenance differs greatly from the defect type distribution during inspection or testing, then it indicates that the inspection or test process is not focusing on the correct types of defects. Instead of using defect information from maintenance, some research uses defect seeding, which is costly and difficult to implement. Defect seeding is an approach attempting to estimate the size of the defect population and the effectiveness of defect detection techniques by deliberately introducing defects into a system.

3.3.1 Software Defect Detection Process Controlling

Software development teams face diverse, even contradictory, requirements; such as tradeoffs between software quality, cost, and marketing environment. Software defect detection has to take the competing requirements into account just as do other parts

of the software development process. It is unrealistic to conduct a software inspection to the extent that all the defects that could be detected by inspection are detected before testing. Similarly, it is unrealistic to conduct a software test to the extent that all the defects that could be detected by testing are detected before releasing the product. It is important to determine: what types of defects should be detected at the inspection stage, what types of defects should be detected at the testing stage, and what types of defects could be left in the system. So how defect detection should be conducted depends on the nature of the software (the software function), user expectations and the marketing environment [67]:

- Software function.

  How critical the software product is to an organization's operation.

- User expectations.

  User may be in urgent need of the software functionality and may be tolerant of some defects.

- Marketing environment.

  How urgent the software product needs be put into market.

Although most software developers attempt to remove all defects in a software product, it is impractical to inspect or to test it until all the defects are detected. No matter how extensive a testing is conducted, it is still possible that there are defects remaining [10, 52]. As illustrated by the test progress S curve in Figure 3.3 [37], it is almost always true that the more time being spent on the defect detection, the more defects will be found. But the number of defects detected per time unit (the rate) changes with time. At first, the rate is low and increases gradually. Then it starts to drop as more and more defects are detected and less and less defects are left in a system. After that point, the defect detection process becomes less and less efficient and has to be stopped even if it is known that not all defects were detected.

As noted in Figure 3.3, usually at the beginning of testing, the efficiency of detecting the defects is relatively low. After a period of time, the testing gradually

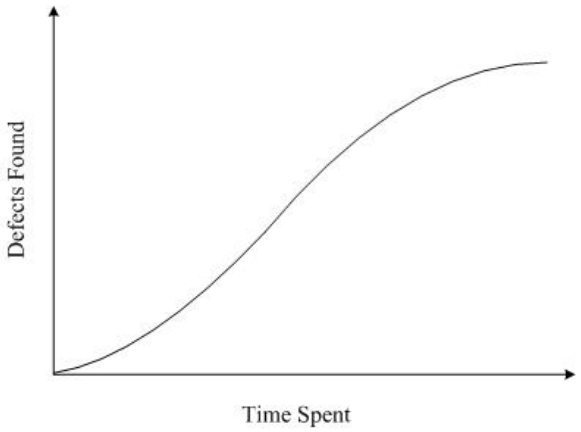

Figure 3.3: Test Progress S Curve Over Time (adapted from [37])

enters into its full function state and the number of defects detected increases quickly. At a later point in time, the defect detection rate becomes lower and lower. So the software defect detection process could be managed by comparing the current rate with a pre-defined limit (the baseline or exit criteria). If the current rate is larger than the baseline, then the current defect detection activity needs to keep on going; otherwise, it can be stopped and the defect detection process moves to the next activity. The baseline for the cutoff point can be determined based on information from previous similar projects in the experience base.

Also, it needs to be noted that a baseline should be established for each type of defect, since not all types of defects have the same impact on the system. As showing 3.1, defect detection techniques are not equally effective and efficient for specific types of defects. Hence classifying defects and treating each type of defect appropriately is a key aspect in controlling, evaluating, or improving the defect detection process. Software defect classification is discussed in the following section, and the usefulness of evaluating defect detection techniques by defect type distribution is presented in Section 3.5 and Section 3.6. The usefulness of improving defect detection techniques by defect trigger distribution is presented in Section 3.7 and Section 3.8.

3.4 The Classification of Software Defects

As stated in the background in Chapter 2, there are several popular software defect classification schemes available in research and industry. In this thesis, software defects will be categorized with the IBM Orthogonal Defect Classification Scheme [12]. The reason for this choice is that it enables in-process feedback to developers, testers, and project managers. With the IBM ODC, a defect is classified by trigger, type, and qualifier. The "defect type" attribute specifies the actual fix that needs to be done for the defect. The "defect trigger" attribute specifies the condition that is necessary for the defect to surface. The "defect qualifier" specifies whether the defect is caused by missing information or incorrect information. The following subsections detail the "defect type" attribute and the "defect trigger" attribute.

3.4.1 Defect Types

To avoid ambiguousness, IBM [12] defined eight possible defect types, and software designers and developers assigned one of these defect types to each defect fixed.

- **Function/Class defect.** An errors Significantly affects the capability of the product/system and causes the product/or system to be unable to fulfill its tasks completely or at all. Usually this defect is caused by the discrepancy between the requirement and design document.
- **Assignment defect.** A variable/structure/object was assigned a wrong value or not assigned at all.
- **Interface.** Errors in communication between two methods, devices, or systems.
- **Checking.** Errors caused by failing to validate the value of a variable or parameter before using it.
- **Timing/serialization.** The necessary sequence to access a shared resource is missing, or the coordination algorithm is wrong.

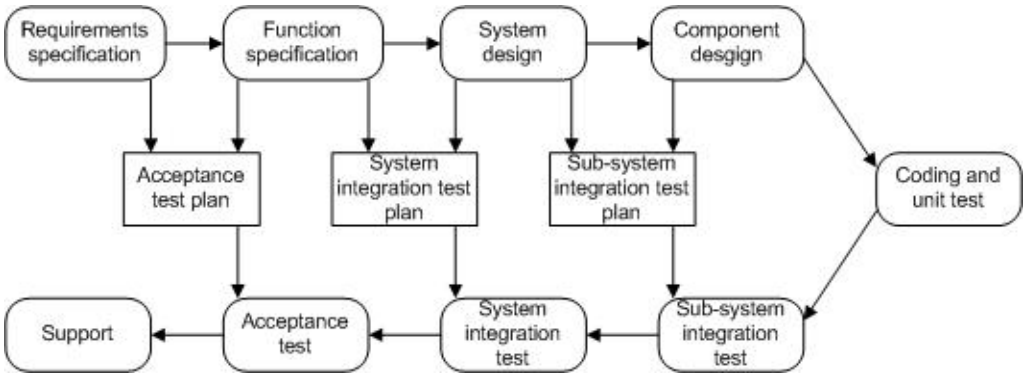

Figure 3.4: V Model for Software Development Process (modified from [46, 67])

- **Build/package/merge.** Errors caused by mistakes in library systems, version control systems, or packaging scripts/tools.

- **Documentation.** The publication provided to help understanding and using of the software was incorrect or incomplete.

- **Algorithm.** The algorithm was inefficient or incorrect.

3.4.2 The Association between Defect Types and Software Development Process

The V-model [46] of software development integrates testing throughout the software life cycle (cf. Figure 3.4). In the V-model, the software development process consists of requirements specification, function specification, system design, detailed design, coding, unit testing, integration testing, system testing, acceptance testing and service. At each stage of the development process, a specific type of defect is more likely to be introduced than at other stages (cf. Table 3.1). For example, if a assignment defect is found (no matter at unit testing stage or at integration testing stage), it would be directly linked to coding. Similarly, an interface error would point to the low-level design. The relationship between defect types and development phases makes it possible to analyze the development process with defect types.

Table 3.1: Defect Type Distribution with Phase (modified from [12])

Defect Type	**Process Association**
Function	Design
Interface	Low-Level Design
Checking	Low-Level Design or Code
Assignment	Code
Timing or Serialization	Low-Level Design
Build or Package or Merge	Library Tools
Documentation	Publications
Algorithm	Low-Level Design

Table 3.2: Process and Defect Type Association (modified from [12])

	Defect Type				
Process	Function	Assignment	Interface	Algorithm	Timing
High-Level Design Inspection	X				
Low-Level Design Inspection			X	X	X
Code Test		X		X	
Unit Test		X		X	
Function/Integration Test	X		X		
System Test					X

3.5 Evaluating the Defect Detection Process Using Defect Type Distribution

Since each defect type tends to be brought in a specific phase, a relatively high percentage of defect types will be detected in the corresponding defect detection activity [12]. This observation makes it possible to build the associations between defect types and defect detection activities (Table 3.2).

For example, the 'function' defect is associated with design and is expected to be detected at both the high level design inspection and also function verification test. The percentage of defects of type 'function' should be high at these two phases. By contrast, low percentage of 'function' defect should be expected before and after

these two stages. The above table thus "describes the profiles of the defect type distribution explicitly" [12]. A deviation means the corresponding defect detection activity is not effective enough. So the defect detection activity can be evaluated by comparing and analyzing the distribution of defect types through the defect detection process. For example, if many logic errors or algorithm errors are detected during the integration testing, this probably means the unit testing is not effective enough. Similarly, if a high number of interface problems are found during system testing or factory acceptance testing, it probably means the integration testing is not well done. If the observed distribution is not as expected, the current defect detection activity need be improved. Also, data gathered from similar projects can be calibrated and tailored to form a baseline for a specific environment [24]. The baseline makes the analysis of the defect type distribution possible and the evaluation of the current defect detection activity can be performed by comparing the current defect type distribution to the baseline.

3.6 Comparison of the Current Defect Type Distribution With the Baseline

Figure 3.5 shows the comparison of the defect type distribution after function testing with the baseline for one of our recent projects. At the function test stage, a large percentage of function defects were expected to be found. However, comparing with the baseline, a relatively small percentage of function errors were found. On the other hand, a large proportion of assignment, checking, algorithm errors were detected, although these defect types should have been detected and eliminated at the unit test stage. The deviation from the baseline demonstrated that the unit test of the project was not effective.

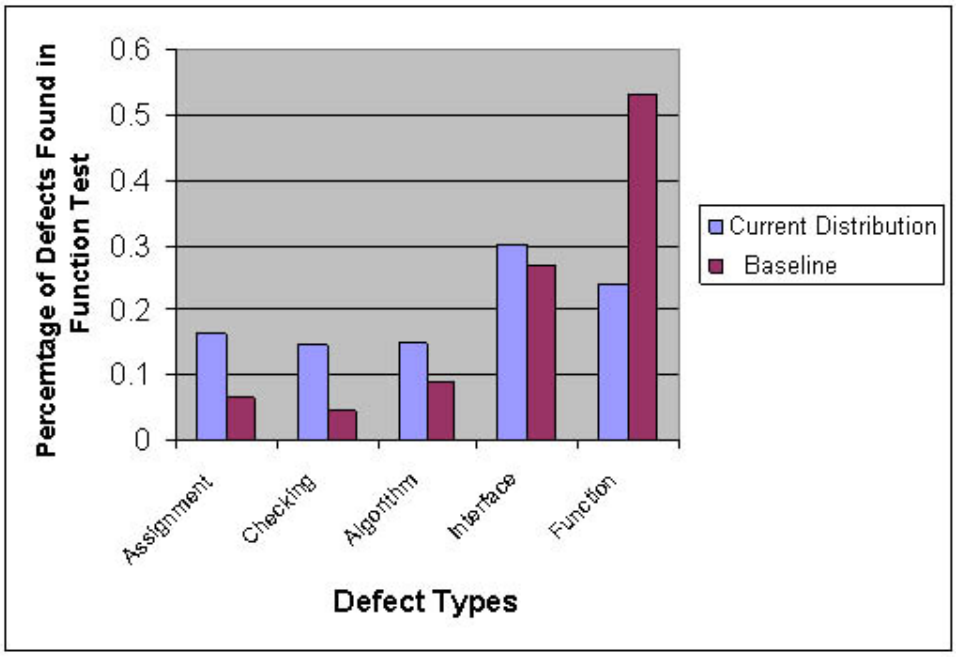

Figure 3.5: Comparison of the Current Defect Type Distribution With the Baseline

3.7 Defect Triggers

In the last section, the defect type attribute was discussed, and it was illustrated that the defect type distribution at different stages could potentially be used to evaluate defect detection techniques. This section addresses another defect attribute, the defect trigger, and it is demonstrated that trigger distribution can be used to improve the defect detection process.

A defect trigger is a condition that leads to a defect being exposed. Defect triggers can be grouped into two categories: inspection triggers and testing triggers.

3.7.1 Defect Triggers in Inspection

Inspection triggers include [12]:

- **Design Conformance.**

  The defect was detected by comparing the design document or code with the

corresponding specification.

- **Understanding Details.**

  The defect was detected by considering the details of the structure and/or operation of a component. Examples include the logic of an algorithm, the side effects of a method, and the calling sequence of two methods.

- **Backward Compatibility.**

  The defect was detected by noticing an incompatibility between the previous versions of the product and the current version under review.

- **Lateral Compatibility.**

  The defect was detected by uncovering an incompatibility between the product or subsystem under review and another product or subsystem with which it needs to communicate.

- **Rare Situation.**

  The defect was detected while considering an uncommon scenario. Such as quitting an operation while in the middle of processing.

- **Document Consistency/Completeness.**

  The defect was detected by uncovering inconsistency or incompleteness in the document.

- **Language Dependencies.**

  The defect was detected while verifying the language-related component(s).

3.7.2 Defect Triggers in Testing

Inspection triggers include [9]:

- **Coverage.**

The defect was detected during unit testing by examining which lines of code are visited (code coverage testing) and/or the ways of getting to each line of code (path coverage testing).

- **Variation.**

  The defect was detected during unit testing by changing the input parameter.

- **Sequencing.**

  The defect was detected during function testing by examining more than one unit one after another and these units do not interface with each other.

- **Interaction.**

  The defect was detected during function testing by examining more than one unit; one of which interfaces with another.

- **Workload/stress.**

  The defect was detected during system testing by changing the workload of the system.

- **Startup/restart.**

  The defect was detected during system testing while restarting the system.

- **Configuration**

  The defect was detected during system testing while changing the system configurations, such as, connecting to the server.

3.8 Discover the Opportunity of Improvement Through the Defect Trigger Distribution

While the defect type provides a mechanism to evaluate the defect detection process by discovering which defect detection activity needs to be improved, the defect trigger correspondingly provides a mechanism to improve the defect detection process by

identifying how the defect detection activity needs to be improved. As we know, different types defect are more likely to be detected by inspectors/testers with the corresponding knowledge/experience. For example, backwards compatibility defects could be detected by a inspector who only has the knowledge of that product, whereas lateral compatibility needs people with experience of both the current product and other related products. Similarly, a test case for coverage in unit testing can be developed by a tester as long as the tester understands that single method or function. To develop a test case for Interaction or configuration, the tester must have extensive knowledge of the functions of the system. After building a chart, listing the defect triggers and the needed skills for the corresponding triggers (Table 3.3), it is possible to infer the weakness of a specific defect detection activity by comparing the trigger distribution with the baseline of the expected trigger distribution. A significantly lower percentage of certain types of triggers would indicate that the inspector/tester is short of the necessary knowledge and the corresponding training should be provided based on Table 3.3.

Table 3.3: The Association Between the Skills and the Defect Triggers Found in Inspection (adapted from [12])

	Knowledge Required			
Triggers	New/Trained	Within Product	Cross Product	Very Experienced
Design Conformance	X	X	X	X
Understanding Details		X	X	X
Backward Compatibility		X		
Lateral Compatibility			X	
Rare Situation				X
Document Consistency		X	X	X
Language Dependencies	X	X		X

As shown in Figure 3.6, by comparing the current defect trigger distribution with the baseline, it is obvious that the triggers which require little experience in this software product are most common: Design Conformance (38.5%) and Document Consistency (29%). On the other hand, the triggers which require extensive experience are a small portion of triggers: Backward Compatibility (6.8%) and Lateral Compatibility (7.2%). The corresponding trigger distribution in the baseline is: Design Conformance (17.5%), Document Consistency (16%), Backward Compatibility (19.8%), and Lateral Compatibility (30.2%). Through the comparison of these two distributions, it can be concluded that the inspection is not effective because the inspectors are short of the experience needed. So a second round of inspection by inspectors who have knowledge of the previous versions of the current product and other related products would be expected to find more backwards compatibility and lateral compatibility defects.

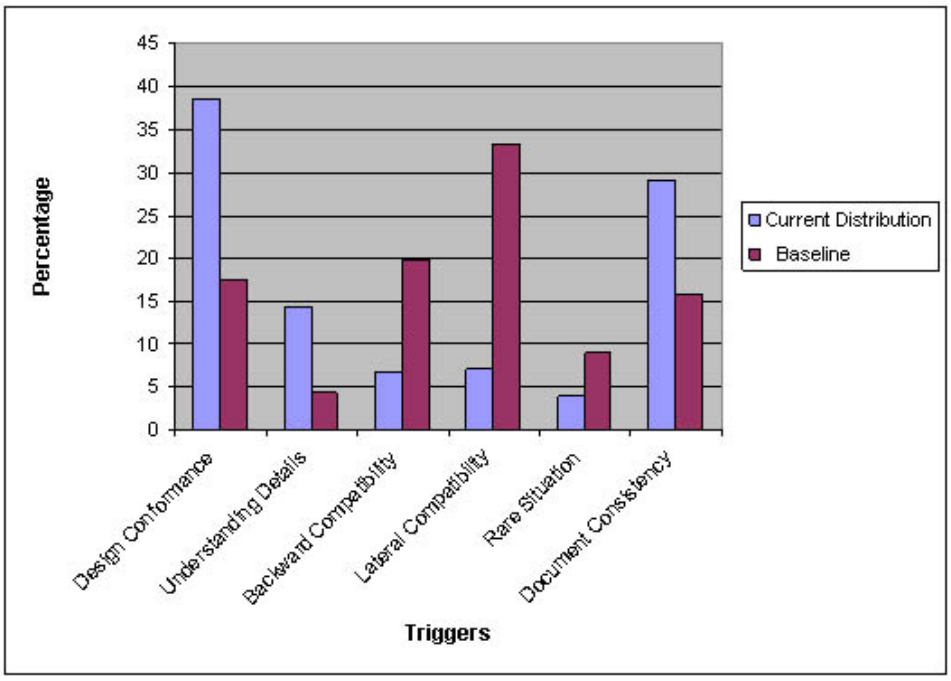

Figure 3.6: The Comparison of Trigger Distribution

3.9 Summary

In this chapter, the concept of and the rationale for the integrated approach to the software defect detection process was introduced. How to control, evaluate, and improve the new approach by collecting and analyzing the defect type and trigger information and comparing the current distribution to the baseline was discussed. In the next chapter, the software architecture for providing software support for this new approach will be presented.

Chapter 4

SOFTWARE ARCHITECTURE OF THE SELF-EVOLVING DEFECT DETECTION PROCESS

In the last chapter, a systematic approach to software defect detection was presented. This new approach can help conduct the software defect detection process by combining different defect detection methods together to achieve the optimized result, can help control the software defect detection process by checking the entrance criteria and the exit criteria, can help evaluate the software defect detection process by comparing the results with a baseline (or historical data), and can help to improve the software defect detection process by discovering the weakness in the current process and the corrective actions that need to be taken.

This chapter presents a software architecture for implementing the self-evolving defect detection (SEDD) process by defining the necessary components, their functionalities, and the relationship between these components for the new approach.

4.1 Major Components of the SEDD Software Architecture

The main concepts of the software architecture come from SPICE [19] and Bootstrap [31], in other words, this software architecture is a specific application of the general concepts in SPICE and Bootstrap to the software defect detection process. SPICE and Bootstrap are process maturity models (not process models) for general

engineering processes (not for a specific process, such as, defect detection), while the software architecture of SEDD is a process model for software defect detection.

According to the requirements in Bootstrap, SEDD defines three process areas (cf. Figure 4.1) (A process area is a set of processes serving the same goal[66]): the key processes (Inspection, Testing, and Maintenance), the supporting processes (Employee training, Automatic tool support, Process changing, Rule and Checklist updating), and the improvement processes. The key processes are the core of SEDD. The supporting processes assist in conducting of the key processes. The improvement processes ensure continuous improvement of the software defect detection process.

Aside from the three process areas, there are two databases: the defect database, which stores the defect data; and, the experience database which is a repository for integrating information from similar projects, the baseline, the goal, and the criteria.

As a whole, these components provide the following functionalities:

- Discover defects.
- Facilitate detect detection technique selection and evaluation.
- Provide information for process control, evaluation, and improvement.
- Enhance the software quality and reduce cost.
- improve employees knowledge and skills.

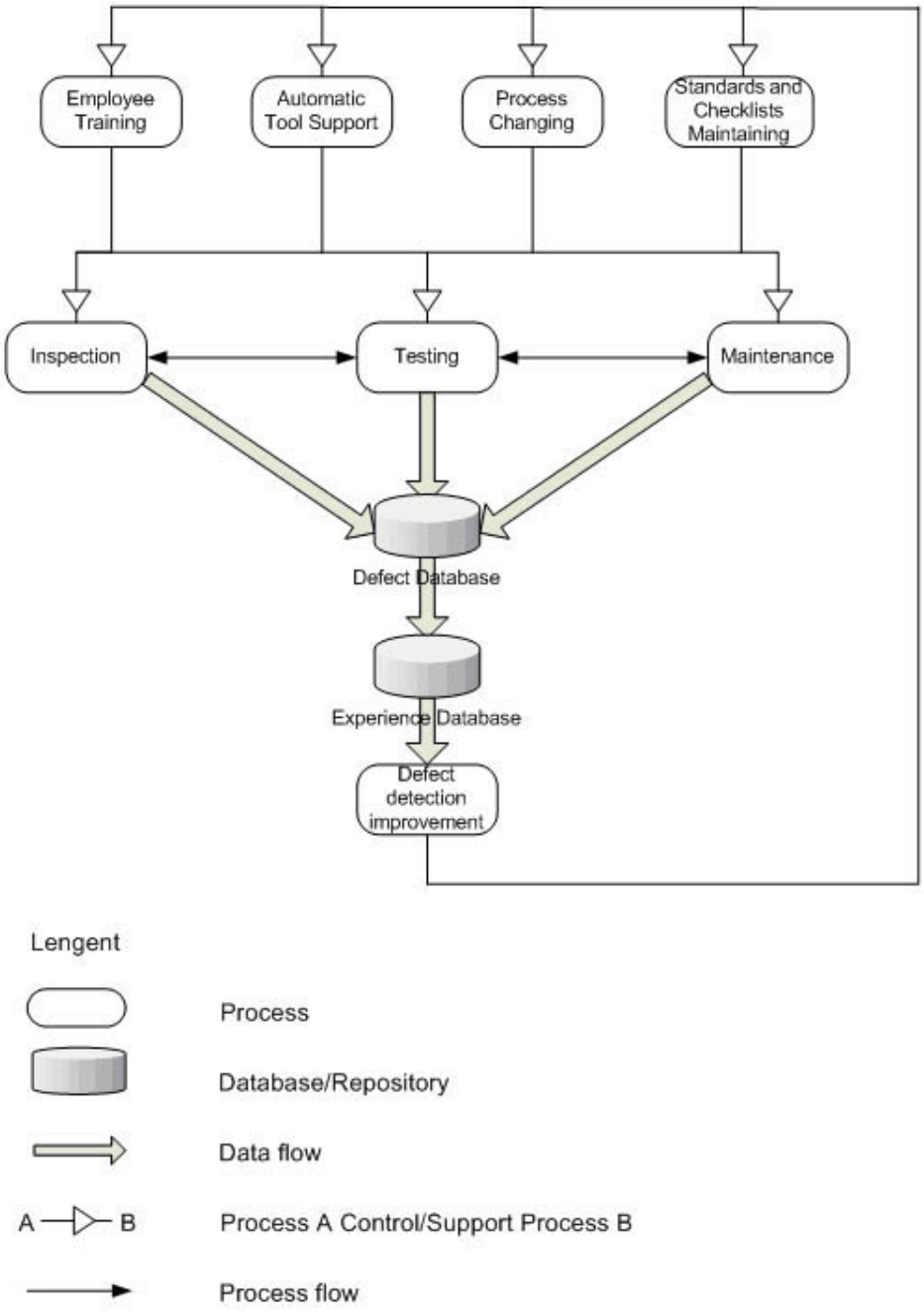

Figure 4.1: The Major Components of the Model

4.2 The Functionalities of the Major Components and the Relationship Between These Components.

4.2.1 Supporting Processes

Supporting processes provide an environment for conducting, controlling, evaluating, and improving the software defect detection process. These processes include: employee training, automatic tool support, process changing, and standards and checklists updating.

Automatic tool support

The function of the automatic tools falls into three categories:

- Support the defect detection process. These tools help to conduct inspection, testing, or maintenance. Any tools used by the inspector, tester, or support staff to execute inspection, testing, or maintenance belongs in this group.

- Collect defect information. These tools help to collect software defects during inspection, testing, and maintenance, and to record the defects for later analysis. According to its functionality, the defect collection tool should be easy to access for all the users (inspectors, testers, and maintainers) and for different locations (in-house and in the field). Therefore, a web-based tool would be a good candidate for the defect collection tool.

- Analyze the defects. After the defects have been collected, users need tools (usually visual tools) to analyze, compare, and present the defects. These tools transform the large volume of raw data into a report, diagram, or chart, helping users understand, analyze, and interpret the data, and eventually helping the users to draw a conclusion or make a decision (Figure 4.2).

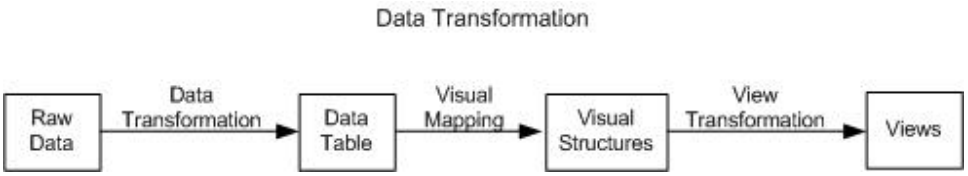

Figure 4.2: The Process to Visualize the Raw Data (modified from [27])

Visual analysis and presentation tools display data in a visual format and help users recognize patterns and trends hidden in the raw data. Without automatic tool support, the raw data would be too hard to understand.

There are lots of visual tools available on the market; they may be as simple as a spread sheet product, or as complicated as the data warehouse products from Cognos [15] and BusinessObjects [8].

Employee training

Employee training provides the employee with the necessary knowledge through certification, tutorials, or courses so that they are qualified to perform their tasks.

Process changing

Process changing adjusts the way inspection, testing, or maintenance is conducted, include adopting a different inspection or testing strategy, using different inspection or testing techniques, and changing the entrance criteria or exit criteria.

Maintaining standards and checklists

The Standards and Checklists Maintaining activity maintains and updates the criteria which an inspection, testing, or maintenance process needs to meet, the rules these processes need to follow, and the checklist these processes need to verify against. To facilitate the conducting of the defect detection process, the following information should be maintained and updated:

- Requirements

- Schedule
- Resources
- Role description
- Checklists
- Rules
- Forms
- Process change order
- Entry criteria
- Exit criteria

4.2.2 The Key Activities of Software Defect Detect Process

Inspection, testing, and maintenance are the essence of the defection detection process. The effectiveness and efficiency of the defect detection process depends on how well these core activities are conducted and controlled.

To have better control over the software defect detection process, inspection, testing, and maintenance must be conducted in a systematic framework. Based on the requirements of SPICE and Bootstrap, the base practices of inspection, testing, and maintenance are defined. These base practices include: entry criteria checking, defect detection technique selecting, executing, defect collecting, defect classification, defect analyzing, and exit criteria checking.

Entry criteria checking: Before starting a practice, the entrance criteria should first be checked to make sure the precondition to execute the practice is mature. Are the software products to be inspected/tested ready? Are the form, checklist, tool, and test cases be prepared? And are the resources needed available?

Defect detection technique selecting: If the entrance criterion is satisfied, the next step should be selecting the proper technique (or a combination of different

techniques) and supporting tools. The selection is based on the requirements of the current project and the information from similar previous projects; such as the time, the cost and the defect detection efficiency from the experience database. Since software defect detection needs to take the economic factor into consideration, the question is not simply which technique can detect more defects than the others, but for a specific type of defect, when (at which stage) and how (using what technique or combination of techniques) it should be dealt with to achieve the best economic result.

Defect collecting: The defect collecting activity involves collecting defects identified during inspection, testing, or maintenance, and entering them into the defect database. In the defect data collection process, attributes, such as defect type and defect trigger, are assigned to each defect. These attributes are very important information to the assessment and improvement of the effectiveness of software inspection and test processes.

Both the set of attributes and the set of values of an attribute are continuously improved by iterative adjustment. At first, it is possible that the set of the attributes and the set of values of an attribute, which come from literature and industry, are not specific enough. But through project to project, this both sets are gradually tailored according to the specific requirements of the organization/department/project, and both sets are constantly enriched and adjusted during inspection, testing, and maintenance.

Defect classification: During the classification process, all defect attributes (as shown in Figure 4.3): the defect type, defect trigger, the impact, the location (where it was found), the phase (when it was found), and the cost (the time spent to detect and fix it) are identified based on the classification scheme. The trigger, the impact, the phase, the location, the effort to detect are decided by the inspector/testor who detected the defect. The defect type and the time to fix it are determined by the software designer or the programmer correcting the defect.

The defect type attribute identifies the actual fix that needs to be done. If the problem is caused by a variable, structure, or a class not initialized properly, then

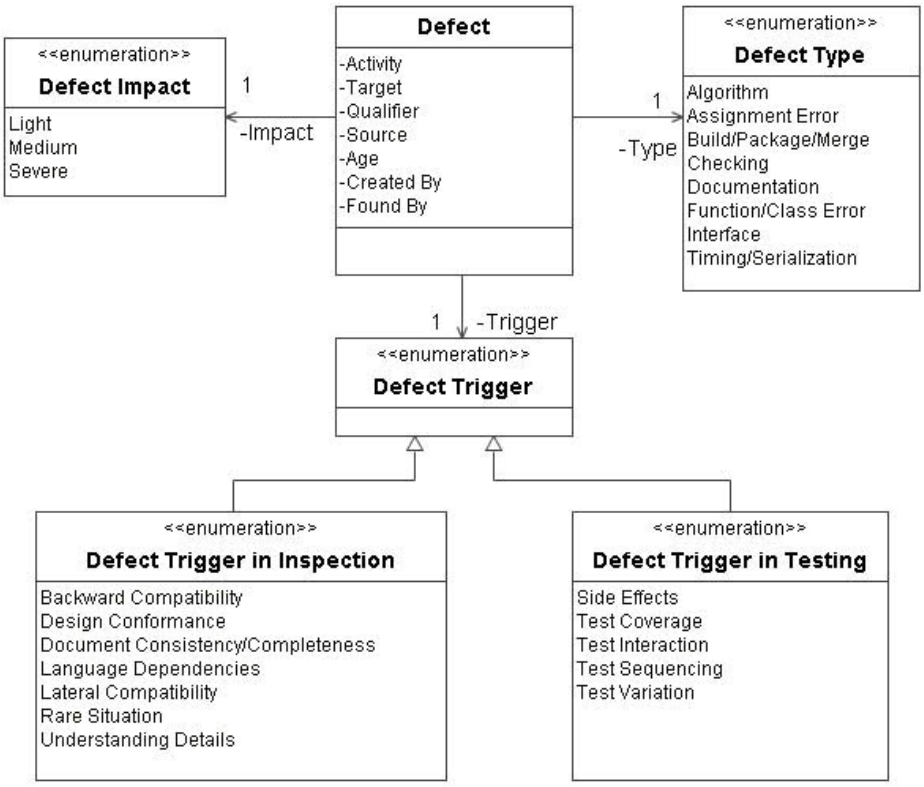

Figure 4.3: Defect Attributes

this defect belongs to the assignment group. If the problem is caused by memory allocation or de-allocation, then it belongs to the memory group. If the problem is caused by using up CPU, then it belongs to the CPU usage category. If it is caused by a concurrent event, then it belongs to the racing category. If it is caused by the interaction of the modules, then it belongs to the interface group. For each of these types, the designer or developer specifies whether the defect is one where information in the artifact is missing, or the information is incorrect.

Defect triggers can help identify the weakness in the software defect detection process. We know that certain types of defects tend to be found by inspectors/testers with certain knowledge or experience. Failing to detect a specific type of defect may indicate that the inspector/tester is short of the specific knowledge and is in need of training in the specific field. Defect triggers can also help building a more balanced inspection or testing team.

As well, other defect attributes can be used to improve the defect detection process. By cross-referencing the defect type with the phase in which the detect was detected and the cost to detect and fix the defect, we can find out what stage is the best to detect that kind of defect to achieve the best cost-benefit result.

Defect analysis: Since each defect type is more likely to be brought in at a specific phase than other phases, the corresponding defect detection activity should discover more defects of that type than other defect detection activities. Deviation from this expected pattern indicates the ineffectiveness of the defect detection activity and the need to improve it [11]. For example, if many logic errors or algorithm errors are detected during integration testing, this probably means that unit testing is not effective enough. Similarly, if a high number of interface problems are found during system testing or factory acceptance, it probably means that integration testing is not well done. When this abnormal situation is found, certain actions need be taken: first the causes of failing to detect the specific type of defect need be identified based on the trigger needed to uncover the defect; for example, was the improper technique used, or the inspector short of the necessary knowledge? Second, corrective actions need be taken to improve the corresponding inspection

or testing activity by using a different technique or a more experienced inspector. In addition, the inspection or testing need be performed again until the defect type pattern conforms with the expected one.

The analysis of the defect detecting activities is based on the classification of the defects collected during the detection process [11]. To analyze these classified defects and evaluate the current activity based on the analysis, the distribution of defect attributes for the current process need be compared with a baseline. The current distribution is obtained by calculating the percentage of each attribute of the defects detected during the current process. The baseline is extracted from the experience base based on data from the similar projects. The baseline is calibrated and tailored according to the specific requirement of the specific project. Also, it is constantly refined project by project by iterative enrichment and adjustment during inspection, test, and maintenance. The resulting baseline would allow for a numerical analysis of the defect attribute distribution. Differences between the current distribution and the baseline form the basis for evaluating a defect detection activity [11]. If the current distribution conforms to the baseline, then it would indicate that the defect detection activity performed normally and it is ready to move to the next step in the software development process. Otherwise, it would imply that the current defect detection activity needs be improved, and the inspection or test needs to continue. Problem reports from customers would also be fed into the defect database, and the inspection and test team could analyze these defects to find flaws in the defect detection processes and improve the process by eliminating the flaws. For example, a large difference of the defect attribute distributions before and after product release would indicate that the defect detection process does not detect the types of defects that really effect the customers.

Exit criteria checking: The last step of the defect detection activity is exit criteria checking. It is almost always true that more inspection/testing will find more defects [11] and it is also always true that software development is constrained by time and money, so the question is not whether we have detected all the defects, but have we got the defects under control. So the process should move to the next

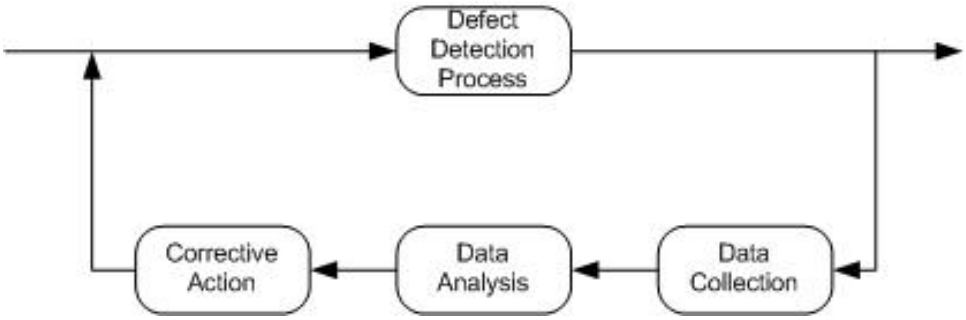

Figure 4.4: Process Improvement

step after the defect attribute distribution conforms to the baseline, and the defect removal ratio satisfies the requirement. At the same time, the experience gained and the lessons learned during the process should be entered into the experience base. For example, did the technique used perform well or not, was the tool used efficient or not, what experience the tester/inspector should have had, and what change needs to be made to the checklists or forms.

4.2.3 Process Improvement Activities

The third important component of the software architecture is the process improvement activity. This activity provides the software defect detection process with a continuous improvement mechanism based on the information from the defect database and the experience base.

In order to improve the performance of the software defect detection process from project to project, the continuous improvement of the processes must be addressed [34]. Such improvement can only come about after identifying the problems in processes by analyzing the metric data obtained from it and taking the corresponding corrective actions to solve the problems (as shown in Figure 4.4). These actions include training employees to improve the inspectors' and testers' technical knowledge and skill, updating of checklists, adopting of new forms and metrics for measurement data, using different inspection or testing techniques, developing better test cases, and performing better scheduling.

4.2.4 Supporting Databases

Besides the activities, there are two databases in the software defect detection process model: the defect database and the experience database. These databases play an indispensable role in the software architecture.

Defect database

All the defects found by inspectors, testers and maintenance staff are stored in the defect database. In addition to the defect type and defect trigger, the following important attributes of a defect should also be stored:

- Project

  Specifying in which project the defect was found enables cross-project comparison, and thus helps evaluate and control of the current defect detection process.

- Phase

  Specifying in which phase the defect was found enables the evaluation of different components of the defect detection process: inspection, testing, and field support. It also makes it possible to discover why the defect was not detected in an earlier inspection or testing phase.

- Created by

  Specifying who created the defect allows an inference to be drawn between the designer or developer and the types of defects. Thus, a customized checklist or test case could be prepared for the designer or developer. Also it helps to identify what kinds of training the designer or developer needs.

- Found by

  Specifying who found the defect makes it possible to find the relationship between the inspector or tester and the type of defects. In other words, who is good at finding which type of defects. The expertise areas of the inspector or tester could by identified.

- Time to find

  Specifying the time spent on finding the defect enables quantitative analysis and evaluation of the different defect detection techniques.

- Time to fix

  Specifying the time spent on fixing the defect allows the defect detection process to be conducted and controlled based on quantitative information and enables the economic analysis of the defect detection process.

Experience database

The experience database is a repository of integrated information regarding the defect detection process of all projects. For the defect detection process, the following information is stored:

- Characteristics of Project

  The scope of the project (the size), the complexity of the project, the type of the project (new or update, web-based or database-based or real-time control system), and the programming language used for the project (procedural language or Objected-Oriented language).

- Schedule

  The total number of days scheduled for defect detecting, the number of days scheduled for inspection, the number of days scheduled for testing, and the number of days anticipated for maintenance.

- Time Spent

  The total number of days actually spent on defect detecting, the number of days spent on inspection, the number of days spent on testing, and the number of days spent on maintenance.

- Resource

  The number of staff who worked on defect detecting, the number of staff who

worked on inspection and who they are, the number of staff who worked on testing and who they are.

- Inspection Strategy and Techniques

  The inspection strategy and techniques used and the result (the number of defects detected at the current stage and the number of defects discovered at a later stage for each type of defect).

- Testing Strategy and Techniques

  The testing strategy and techniques used and the result (the number of defect detected and the number of defect remained for each type of defect).

The above information can help to make a schedule, determine the composition of an inspection and testing team, and determine an inspection and testing technique for a future project.

Other than the information for a specific defect detection process, the experience database also contains information which can be used to get answers on some important questions. Some examples are: Which inspector or tester is good at detecting which type of defect? Which inspection or testing technique is good at detecting which type of defect? Which inspection technique or combination of inspection techniques is more effective or efficient than others? Which testing technique or combination of testing techniques is more effective or efficient than others? Which combination of inspection and testing technique is more effective or efficient than others? It also useful to control and improve the defect detection process by establishing and improving the entry criteria and exit criteria.

4.3 Summary

In this chapter, a software architecture was proposed to implement the systematic approach to the software defect detection process. The major components of the software architecture were introduced and their functionalities were discussed. These components make it possible for the software defect detection process to be

conducted, controlled, evaluated, and improved based on the quantity analysis of the data collected during the software defect detection process. In the next chapter, a prototype to implement the software architecture will be presented.

Chapter 5

PROTOTYPE IMPLEMENTATION OF THE SELF-EVOLVING DEFECT DETECTION PROCESS

In the last chapter, a software architecture was introduced to implement the self-evolving software defect detection process. The major components of the model, their functionalities, and the relationship between these components were discussed.

This chapter presents a prototype for the self-evolving software defect detection management system which implements the model discussed in the previous chapter.

5.1 The Prototype and Its Functionalities

To demonstrate the applicability of the self-evolving software defect detection process approach, a prototype was built. The prototype helps conduct, control, evaluate, and improve the software defect detection process by providing support for defect collection, defect analysis, defect detection process analysis, and standards and checklists maintenance and upgrading. The functionalities are provided by the defect management subsystem, the defect analysis subsystem, and the defect detection process analysis subsystem. They are supported by the standards, rules, and checklists maintenance subsystem (as shown in Figure 5.1).

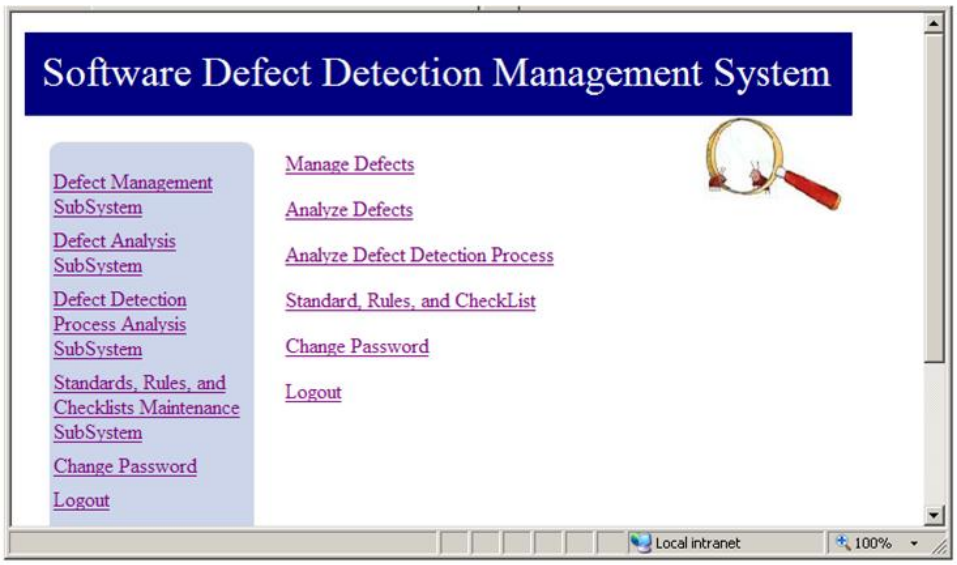

Figure 5.1: The Major Components of the Prototype of a Self-Evolving Software Defect Detection Process Management System

5.1.1 Defect Management Sub-System

The defect management sub-system enables the user to record a defect into the system, search for the defects based on specific criteria, update a defect, and get a list of all unclosed defects (as showed in Figure 5.2).

Record a defect into the system

To make the defect data useful for later analysis, defects must be recorded into the system in a systematic way. The main attributes of the defect must be clearly defined to avoid ambiguities when the user records a defect into the system, and all the attributes must be captured before the defect is closed.

Based on the Orthogonal Defect Classification from IBM [9] and our special needs for the control, evaluation, and improvement of the defect detection process, the following attributes of defects will be captured and entered into the system (Figure 5.3):

- Activity

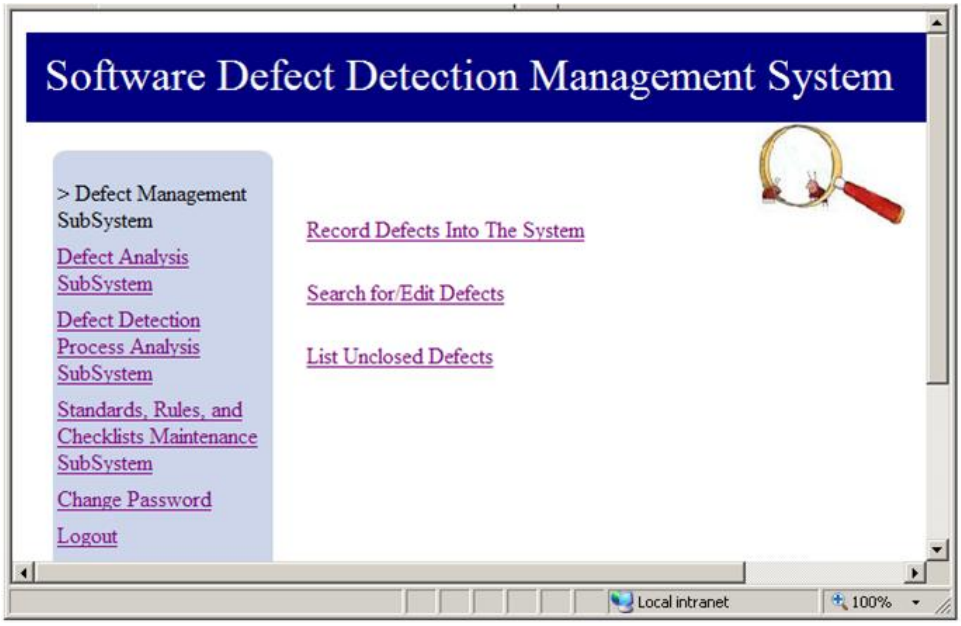

Figure 5.2: The Defect Management System

The practice being performed when the defect was detected, such as unit test, integration test, and maintenance.

- Trigger

  The necessary condition for the defect to surface.

- Impact

  The effect on the customer caused by a defect.

- Target

  The object that was fixed, such as requirements document, design document, and code.

- Defect type

  The actual nature of the fix made.

- Qualifier

  The entity was missing, incorrect, or irrelevant.

- Source

  The origin of the defect discovered: in-house code, a library, or code from third party.

- Age

  Age specifies the history of the defect detected; for example, the defect was found in new, old, re-written, or re-fixed code.

- Created by

  Specifying who created the defect allows an inference to be drawn between the designer or developer and the types of defects. Thus a customized checklist or test case could be prepared for the designer or developer. Also it helps to identify what kinds of training the designer or developer needs.

- Found by

  Specifying who found the defect makes it possible to find the relationship between the inspector or tester and the type of defects. In other words, it provides information on who is good at finding a specific type of defects. Also, the expertise areas of the inspector or tester can by identified.

- Technique used

  Specify which inspection or testing technique was used.

- Time to find

  Specifying the time spent on finding the defect enables the quantitative analysis, and evaluation of the different defect detection techniques.

- Time to fix

  Specifying the time spent on fixing the defect allows the defect detection process to be conducted and controlled based on quantitative information and enables the economic analysis of the defect detection process.

When a defect is first detected, the activity, trigger, impact, found by, and time to find are captured and entered into the defect database. When the defect has been

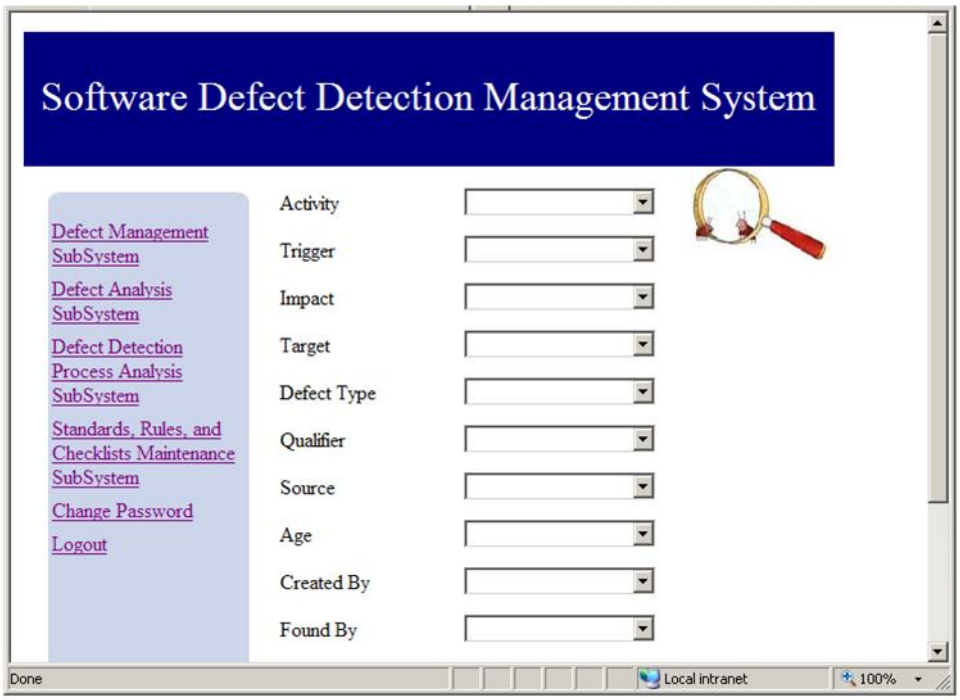

Figure 5.3: The Attributes of a Defect Recorded Into the System

fixed, the target, defect type, qualifier, source, and age, created by, and time to fix are entered.

Assign values to an attribute of a defect

To avoid arbitrariness and ambiguities and to make future mathematical analysis and modeling possible, instead of letting the user enter free text, a list of choices is provided for the user to pick from for the following defect attributes:

Activity: The choices for activity are function specification review, design review, unit testing, integration testing, and system testing (Figure 5.4).

Defect Trigger: A defect trigger is a condition that leads to a defect being exposed. Defect triggers can be grouped into two categories: inspection triggers and testing triggers. The choices in the dropdown list change depending on the phase (inspection or testing) selected. As shown in Figure 5.5, inspection triggers [12] include:

- **Design Conformance**

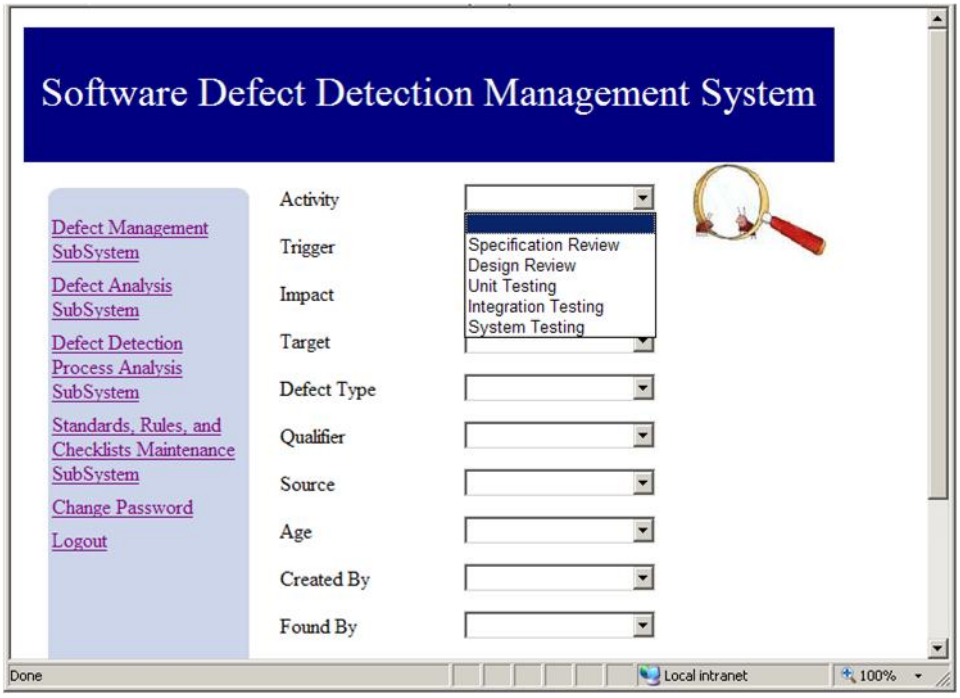

Figure 5.4: The Possible Values for Activity Attribute

The defect was detected by comparing the design document or code with the corresponding specification.

- **Understanding Details**

  The defect was detected by considering the details of the structure and/or operation of a component; for example, the logic of an algorithm, the side effects of a method, and the calling sequence of two methods.

- **Backward Compatibility**

  The defect was detected by noticing an incompatibility between the previous versions of the product and the current version under review.

- **Lateral Compatibility**

  The defect was detected by uncovering an incompatibility between the product or subsystem under review and another product or subsystem with which it needs to communicate.

- **Rare Situation**

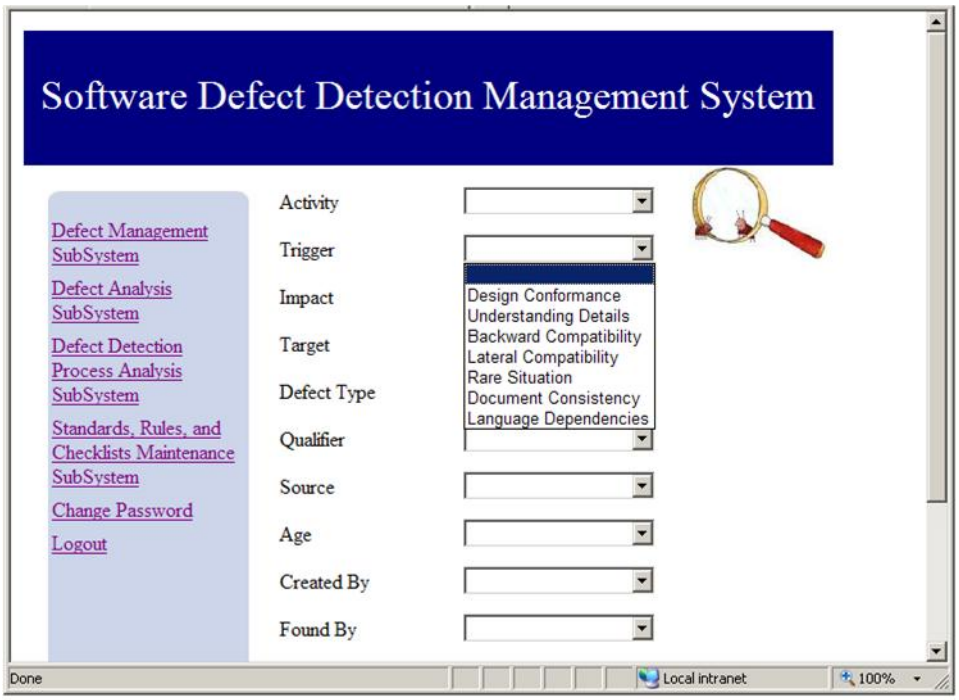

Figure 5.5: The Possible Values for Inspection Trigger

The defect was detected while considering an uncommon scenario; for example, quitting an operation while in the middle of processing.

- **Document Consistency/Completeness**

  The defect was detected by uncovering inconsistency or incompleteness in the document.

- **Language Dependencies**

  The defect was detected while verifying the language-related component(s).

Defect triggers in testing [9] include:

- **Test Coverage.** The defect was detected during unit testing by examining which lines of code are visited (code coverage testing) or/and the ways of getting to each line of code (path coverage testing).

- **Test Variation.** The defect was detected during unit testing by changing an input parameter.

- **Test Sequencing.** The defect was detected during function testing by examining more than one unit, one after another and these units do not interface with each other.

- **Test Interaction.** The defect was detected during function testing by examining more than one unit; one of which interfaces with another.

- **Workload/Stress.** The defect was detected during system testing by changing the workload of the system.

- **Startup/Restart.** The defect was detected during system testing while restarting the system.

- **Configuration.** The defect was detected during system testing while changing the system configurations; for example, the connection to the server.

Impact

The choice for impact include: light, medium, and severe.

Target

The choice for target includes function specification document, software design document, and code.

Defect Types

As shown in Figure 5.6, the possible selections for defect type include function/class error, assignment error, interface, checking, timing/serialization, build/package/merge, documentation, and algorithm.

- **Function/Class error:** Significantly affects the capability of the product/system and causes the product/or system to be unable to fulfill its tasks completely or at all. Usually this defect is caused by the discrepancy between the requirement and design document.

- **Assignment error:** A variable/structure/object was assigned a wrong value or not assigned at all.

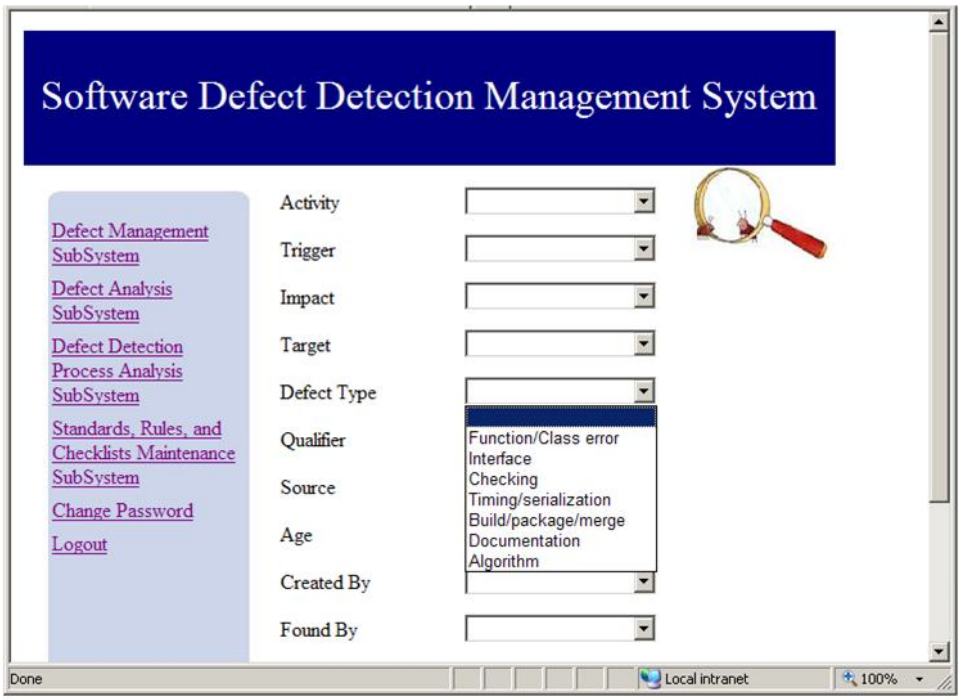

Figure 5.6: The Possible Values for Defect Type

- **Interface:** Errors in communication between two methods, devices, or systems.
- **Checking:** Errors caused by failing to validate the value of a variable or parameter before using it.
- **Timing/serialization:** The necessary sequence to access shared resource is missing, or the coordination algorithm is wrong.
- **Build/package/merge:** Errors caused by mistakes in library systems, version control systems, or packaging scripts/tools.
- **Documentation:** The publication provided to help understanding and using of the software was incorrect or incomplete.
- **Algorithm:** The algorithm was inefficient or incorrect.

Qualifier: Missing or incorrect code/information.

Source: Design documents, code, reused from a library, or ported from one platform to another.

Age: New, old (base), rewritten, and re-fixed code.

The other attributes, created by, detected by, time to find, time to fix, and projects are easy to figure out by their name and are not likely to cause ambiguity, so they are not discussed here.

Search for/Edit defects

This module provides the functions for the user to search for or edit defects based on project, created by, or detected by.

List unclosed defects

This module provides the functions for the user to get a list of all unclosed defects in the system or the unclosed defects for a specific project. With this functionality, the user can easily track the status of the defects.

Email notification

Other than the above functionalities, the defect management subsystem also contains an email notification function to help tracking the status of the defect. Whenever a defect is entered in the system, an email is sent out the corresponding project manager. After the manager assigns the defect to a member to fix, an email is sent the member. After the defect is fixed, an email is sent out to both the person who entered the defect and the project manager.

5.1.2 Defect Analysis Sub-System

Defect Analysis subsystem helps the user analyze the defects by providing a visual representation of the defect data from a different point of view: defect number distribution over creators, defect number distribution over target, defect number

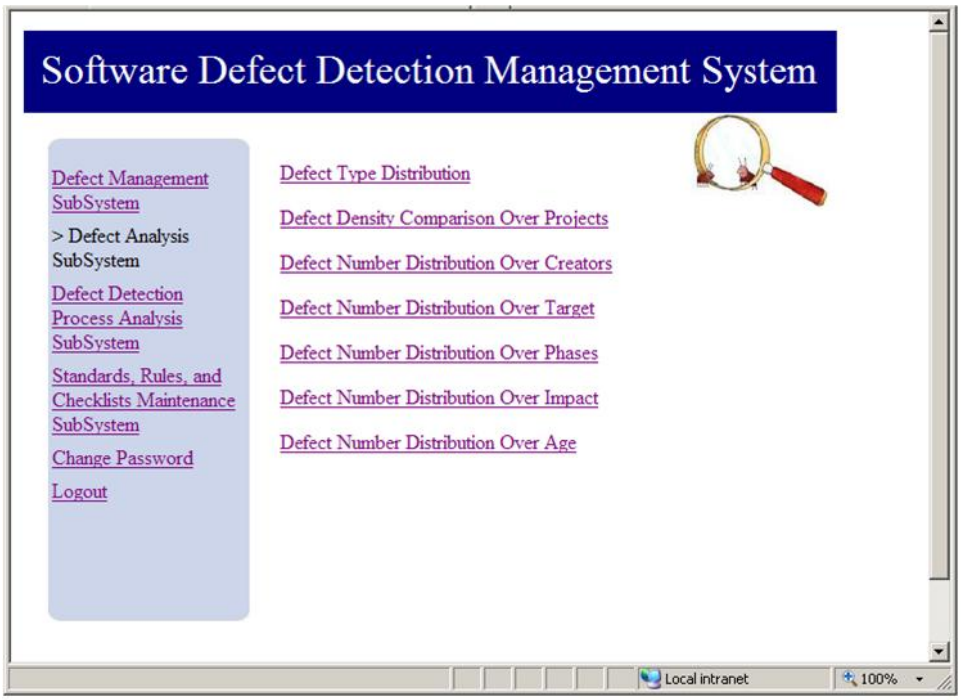

Figure 5.7: The Defect Analysis Subsystem

distribution over phases, defect number distribution over impact, and defect number distribution over age (Figure 5.7).

5.1.3 Defect Detection Process Analysis Subsystem

As shown in Figure 5.8, the defect detection process analysis subsystem enables the user to analyze and evaluate the defect detection process by providing graphical representation of the data related to the process from different perspectives: defect types distribution over activity, defect types distribution over founder, defect triggers distribution over founder, defect types distribution over technique, defect removal effectiveness comparison over techniques, and defect removal efficiency comparison over techniques.

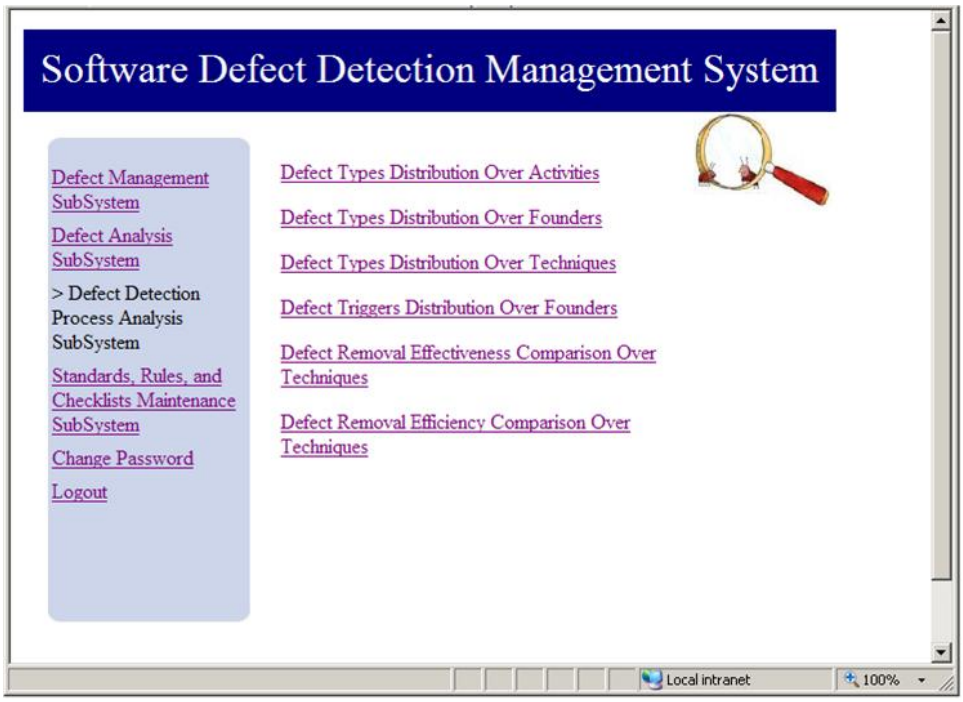

Figure 5.8: The Defect Detection Process Analysis Subsystem

5.2 Summary

In this chapter, a prototype for the self-evolving software defect detection management system was presented to help collect, classify, analyze the defects and to facilitate identifying the problems in the existing defect detection process and improving the process.

Chapter 6

CASE STUDY USING THE SELF-EVOLVING DEFECT DETECTION PROCESS

This chapter presents a case study of the self-evolving software defect detection process approach. It begins with a brief case history and sets the objectives for the case study. Then the quantitative defect data is presented followed by a step by step detailed analysis. After the root cause of the problem in the defect detection process is identified, the corrective action is recommended and the validation of the new approach is demonstrated.

6.1 Case History

The case study was performed at a medium-size company that was established over fifty years ago. The information technology department of the company has about a dozen employees with very different educational backgrounds and industrial experiences. Their education ranges from a one-year diploma to a Ph.D. degree. Their experience level ranges from fresh-out-school to over twenty years of industry experience. Their projects consist of two categories: updating old systems and developing new systems. The old systems were developed on a mainframe. Most of the projects developed in recent years were built on a Microsoft platform: Windows 2000/2003 operating system, Exchange web server, SQL 2000/2005 database server, and Microsoft languages (VB, VB.Net, and C#). The complexity of the projects varies greatly: from a one week project for a single person to several months for eight people.

The development of the projects started with analysis followed by design, coding, testing and deployment. First, the system analyst scheduled a requirements meeting with the end users. At the meeting the systems analyst asked and documented the requirements of the end users. After the meeting, the systems analyst sent the requirements document to the end users to confirm the requirements. The designer started the design based on the requirements. The design inspection was conducted after the design document was completed. Programmers developed the code based on the design and testers tested the code based on the requirements. Finally, the system was deployed after passing the unit, function, and system tests.

The project being studied is a new project which enables the customers to buy our policy (product) online based on the requirements from the policy development department and the marketing department.

The project is a typical modern multi-tier web application with a presentation layer, a business logic layer, a data access layer, and a data storage and management layer. The presentation layer gathers user input and then provides it to the business logic layer, where it can be validated, processed, or otherwise manipulated. The presentation layer then responds to the user by displaying the results of its interaction with the business logic layer. The business logic layer includes all the business rules, data validation, manipulation, processing and security for the application. The data access layer interacts with the data management layer to retrieve, update and remove information. The data access layer doesn't actually manage or store the data; it merely provides an interface between the business logic and database. The data storage and management layer handles the physical creation, retrieval, update, and deletion of data. It was developed and deployed using pure Microsoft technologies: developed in C# and deployed on Microsoft web server, application server, and SQL server 2000.

6.2 Case Study Objective

The primary objective of this case study is to find out if the self-evolving software defect detection process can help improve the defect detection process by giving us a clearer understanding of the current process: how well it performed, what the major problem is, where it needs to be improved, and what corrective action need be taken.

6.3 Data and Analysis

Like the majority of the software projects in the information technology industry, most of our projects were delivered over-budget, behind schedule, and with poor quality. The main reason for this situation is that there is so much rework and maintenance needed to be done to fix the defects found in production which escaped inspection and testing. To change this situation and improve the defect detection process, the root problem in the defect detection process needs to be identified.

Based on this requirement, the defects detected during inspection, testing, and maintenance were collected, classified, and analyzed using the new systematic approach to the software defect detection process.

To get an idea on how each defect detection activity performed, defects detected in all the phases are shown in Figure 6.1.

In Figure 6.1, it is obvious that the percentage of defects that escaped inspection and testing and eventually leaked to production is very high (over 38%). To find what caused this unwanted situation, further analysis of the defects found in production was performed. The results are illustrated in Figure 6.2.

From Figure 6.2 we can see that the dominant defect type is Function. From Table 3.2 (in Section 3.4), we know that function defects in production means High-Level Design Inspection and/or Function Testing did not performed well, and need to be improved. To further investigate, the Source attribute of the function defects are illustrated in Figure 6.3.

From Figure 6.3, we can see that most of the function defects were in the design

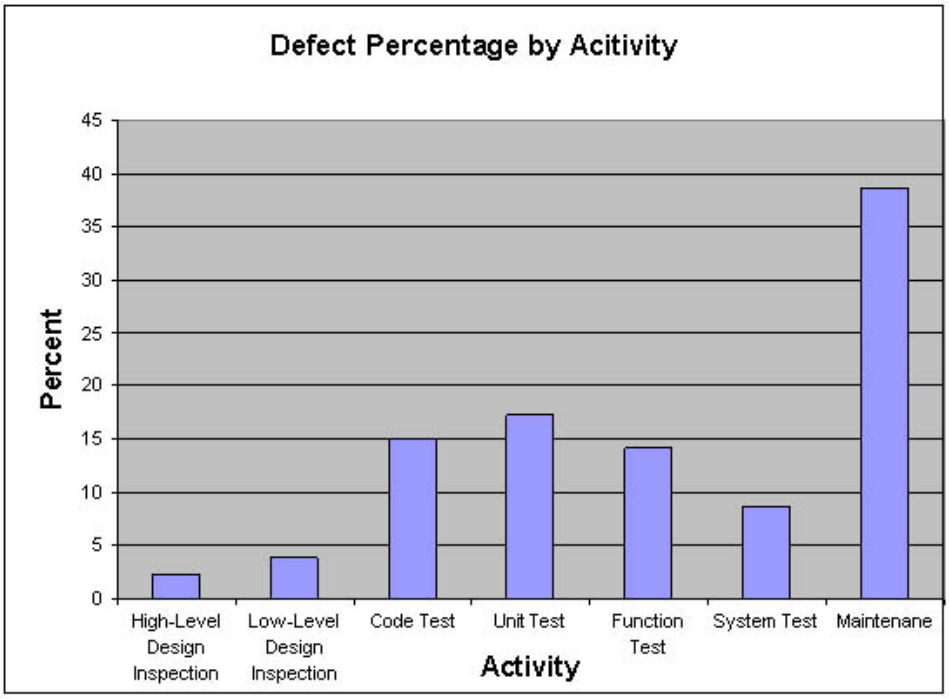

Figure 6.1: Defect Distribution over Defect Detection Activities

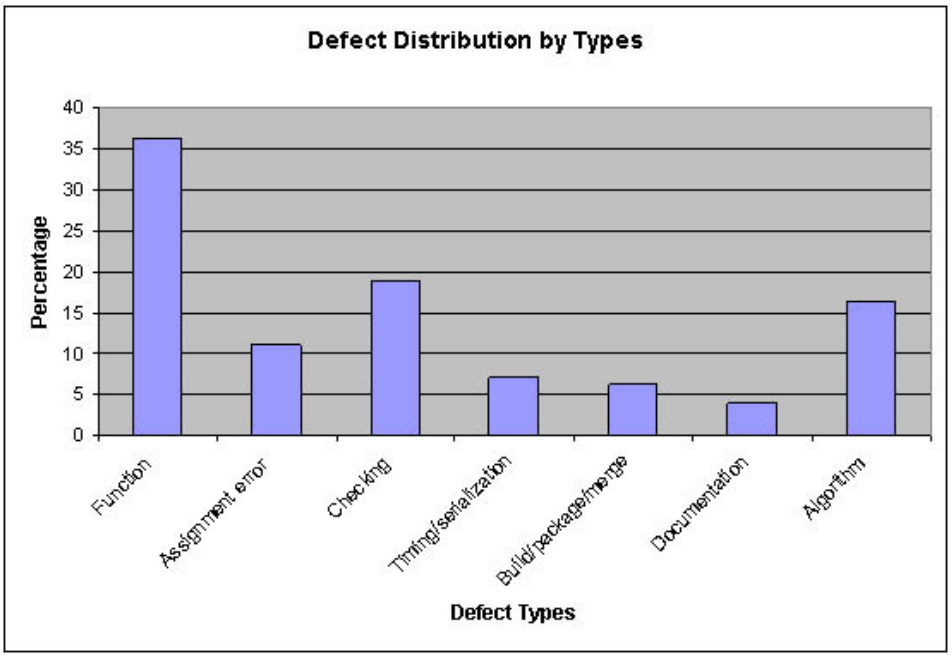

Figure 6.2: Defect Distribution over Defect Detection Types

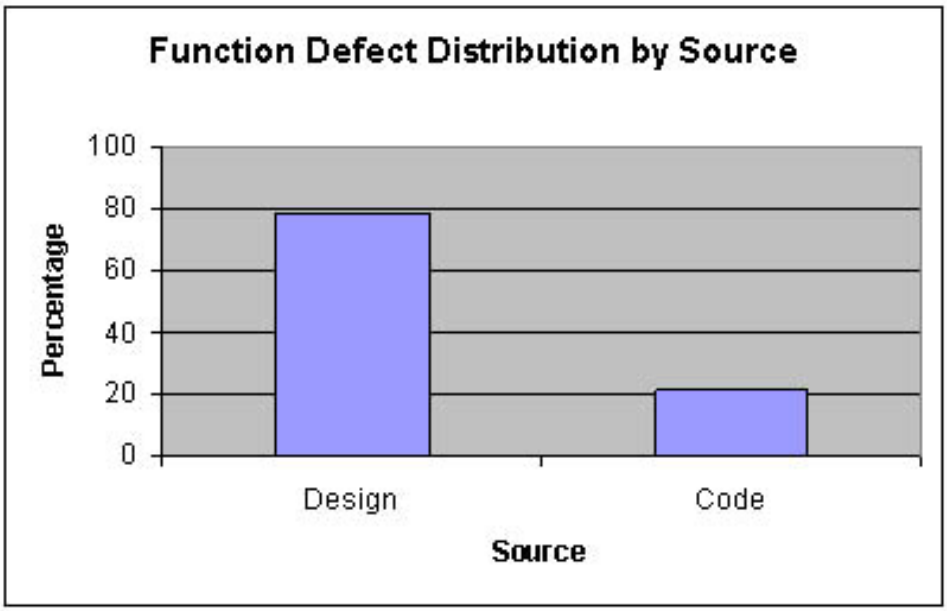

Figure 6.3: Defect Distribution over Defect Detection Sources

and that means both the design and design inspection process need to be improved. To further investigate how these two processes should be improved, the function defects were analyzed by Qualifier (missing or incorrect) as demonstrated in Figure 6.4.

From Figure 6.4, we can see that the majority of the function defects are a result of missing functionality (over 84%). The missing functionality occurred during design and was not captured with design inspection. After we found the cause of the problem, it's time to review the existing design and design inspection process.

6.4 The Existing Design and Design Inspection Process

The existing design process started with the requirement meeting called by the system analyst. At the meeting, the system analyst asked and documented the requirements of the end users. After the meeting the system analyst sent the requirement document to the end users to confirm the requirements. Then the designer started

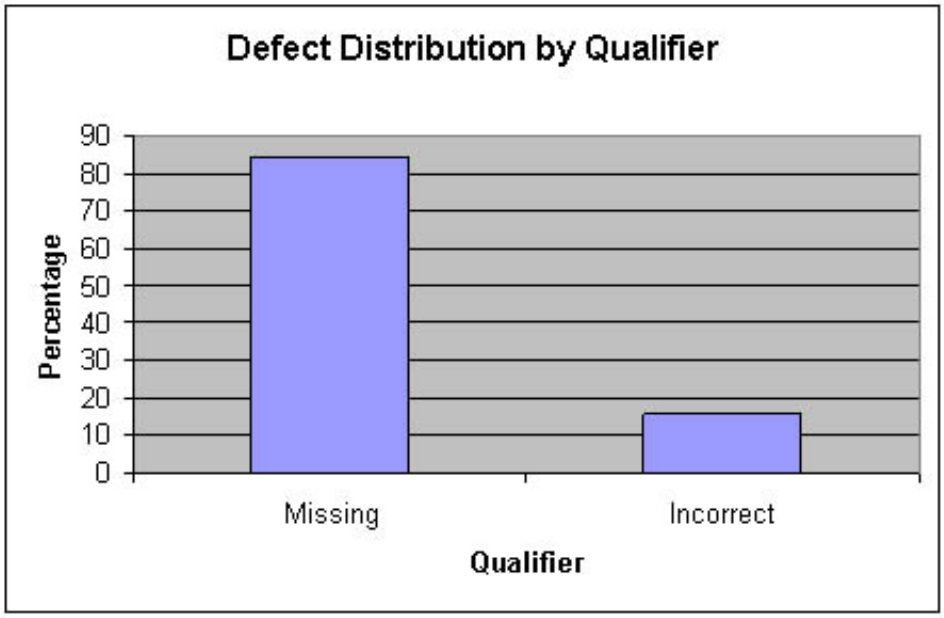

Figure 6.4: Defect Distribution over Defect Detection Qualifier

to design based on the requirements. The design inspection started after the design document was done.

6.5 Problems Identified in the Existing Process Flow

The existing design and design review process had the following problems:

- The users did not tell the system analyst all of their requirements.

  Since the system analyst scheduled the meeting at whatever time resources were available, the end users did not have a chance to think of what they really needed before the requirements meeting, and therefore they were unable to let the system analyst know all of their requirements.

- There often were misunderstandings between the end users and the system analyst.

Since the system analyst and the end user talk a different language, end users do not understand many technical terms and system analysts usually are not very familiar with business terms. Quite often there are discrepancies between what end users want and what they get.

- The design review was based on the system analyst's understanding of the requirements.

  The design review was based on the requirement document which was documented by the same person. So the requirement document and the design document may agree with each other, but the design document does not comply with the users requirements.

- The design review was conducted with the method the reviewer preferred and from the technical persons point of view.

6.6 Corrective Actions to the Existing Design and Design Inspection Process

Based on the problems identified in the current design and the design inspection process, the followed corrective actions were recommended and taken:

- The system analyst must schedule the requirements meeting at least 48 hours before the meeting time so that the end users have time to think about what they really need.

- Each end user must document what she/he needs and present the document to the system analyst before or at the requirement meeting, instead of the system analyst trying to understand and write down a user's requirements while the user is talking.

- After the requirement meeting, the system analyst summarizes the requirements from different end users and documents and presents a function specification document to the end users instead of the requirements document.

- Design should not be started until end users are satisfied with and signed off the function specification document.
- The design review should be based on the function specification document instead of the requirements document.
- Use the Perspective-Based inspection technique instead of an arbitrary technique for design inspection so that each inspector takes a different point of view, not only from the system analyst's point of view, but also from the developers and the users point of view as well.

6.7 Results from the New Approach to the Software Defect Detection Process

6.7.1 Improvement after Implementing the Corrective Actions

To find out the improvement (if there is any) after implementing the corrective actions, the average percentage of Function defects before and after implementing the corrective actions were compared. As we know, the longer a product is in use, the more defects are likely to be found. So only the defects detected in the first six months after release were taken into account. Before implementing the corrective actions, over a hundred projects were completed. Of these projects, relevant defect detection information was only available from 22 projects that were completed relatively recently. The average percentage of 'Function' defects before implementing the corrective actions was derived from these 22 projects. Out of the projects developed after implementing the corrective actions, 14 projects have been in use for over six months. So the average percentage of 'Function' defects after implementing the corrective actions was derived from these 14 projects.

As shown in Figure 6.5, after implementing the corrective actions, the percentage of the 'Function' defects dropped from 38.6% to 18.8%. A more important improve-

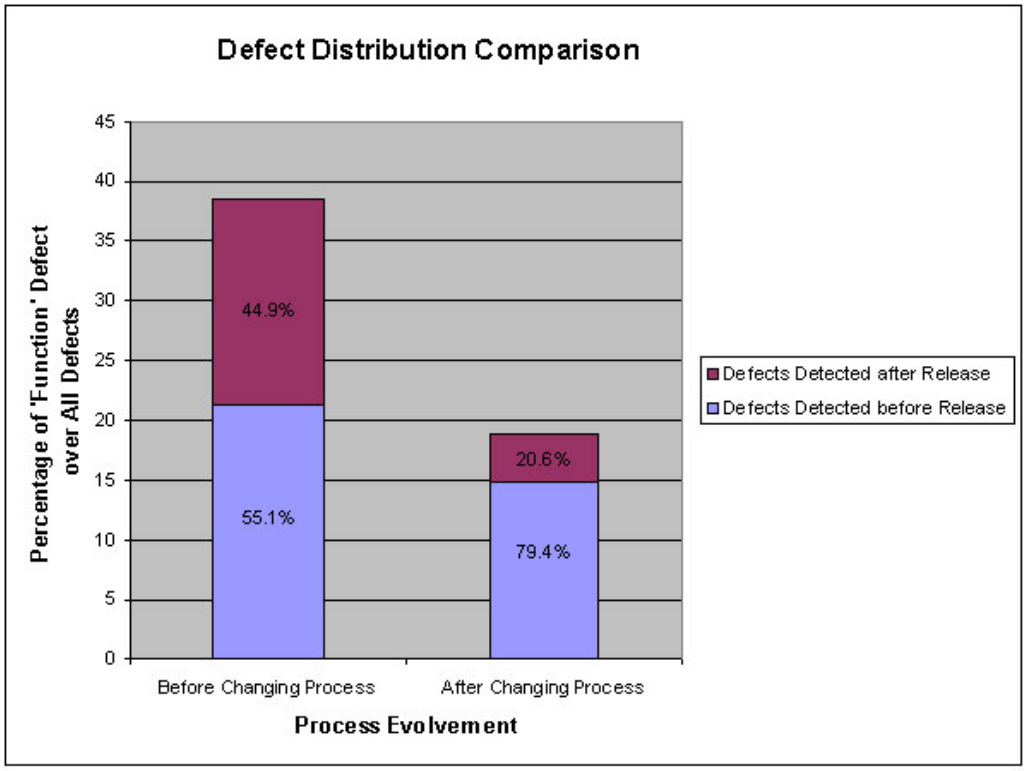

Figure 6.5: 'Function' Defect Comparison

ment is that, percentage of the 'Function' defects detected after release dropped from 44.9% to 20.6%. Detecting more 'Function' defects in the earlier stages may mean that even more savings is realized from the defect detection process, since it is typically the case that removing a defect at an earlier stage costs much less than removing it at a later stage.

6.7.2 Improvement after Implementing the New Approach to the Defect Detection Process

Since the new approach to the defect detection process was implemented, fourteen projects have used it. These projects are very different in terms of the languages (procedure language VB 6 and Object-Oriented language VB .Net), the architectures (client-server and multi-tier), the databases (as simple as Access and as complicated as SQL Server 2005), the resources (new graduates from school and seniors with

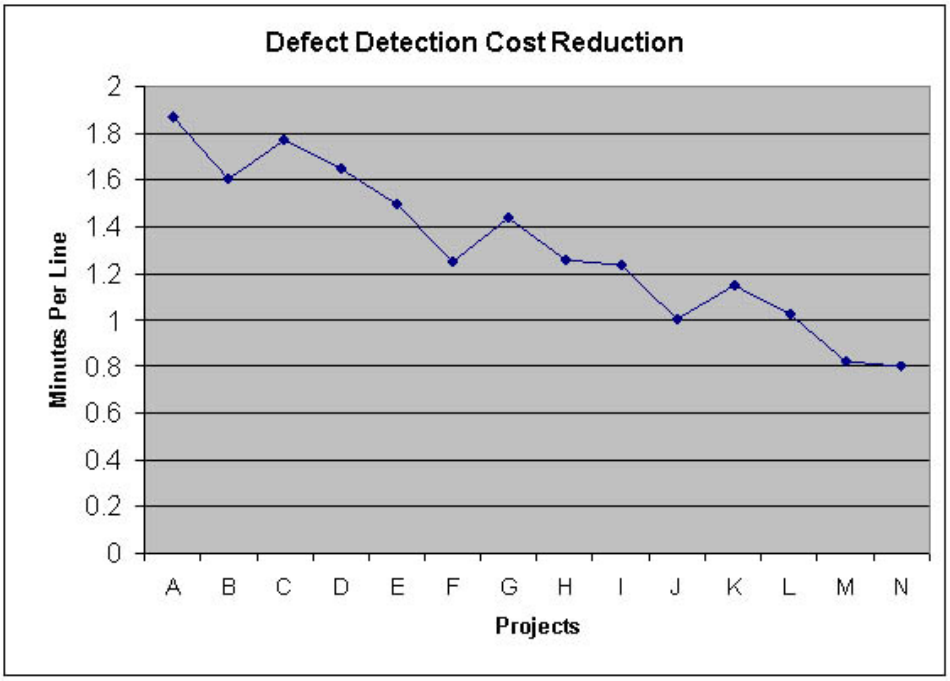

Figure 6.6: Defect Detection Cost Reduction Through Projects

over eighteen years experience), the complexity (from one week for a single person to several months for eight people), and the characteristics (adding new functionalities to an old system, fixing the bugs in an old system, and developing a new system). With only these fourteen projects and the large variations, it is too early to draw a statistical conclusion on what was improved as experience was gained. The cost of defect detection has dropped dramatically, although there are some fluctuations. From Figure 6.6, we can see that the time spent on defect detection has decreased from 1.87 minutes per line to 0.80 minutes per line. It is unlikely that all the improvements directly come from the new approach to the defect detection process. For example, it may be the case that the programmers, the inspectors, and/or the testers pay more attention to their work now that their user IDs are being logged when they register a defect with the system. However, the improvement in the defect detection process using the new approach is encouraging.

6.8 Benefits

The self-evolving software defect detection process approach has the following advantages:

- Recording and classifying the defects in a systematic way to make future analysis possible.
- Helping analyze and evaluate the defect detection process by providing a visual representation of the defects from different perspectives.
- Identifying the root cause of the problem (The Design Document) through the step by step analysis of the defect attributes.
- Helping find out the weakness in the existing defect detection process (Design Review) by analyzing the cross-referencing relationship between phase and type attribute.
- Making it possible to continuously improve the defect detection process by identifying the ineffectiveness of a technique currently used and providing rationale for choosing a different technique.
- Helping iteratively enrich the experience base by adding new findings to it from process to process. For example, perspective-based reading was determined to be more effective than general reading in finding the missing functions in the design document.

6.9 Summary

In this chapter, based on the prototype implementation of SEDD presented in the last Chapter, a case study was performed validating the new approach to the defect detection process by demonstrating how the new approach can help identify what

the major defect is, which defect detection process failed to detect these defects, and what actions need be taken to improve the defect detection process.

Chapter 7

CONTRIBUTIONS, CONCLUSIONS, AND FUTURE WORK

7.1 Thesis Summary

This research investigated the software defect detection process to address: how to conduct the process better, how to evaluate and control the process better, and how to continuously improve the process. The main goals of the thesis are: (1) to propose a self-evolving software defect detection process approach; (2) to present a software architecture for implementing this systematic approach; (3) to build a prototype to partially implement this new approach; and (4) to perform a case study to evaluate the approach.

7.2 Contributions

The contributions of this thesis include the following: First, the new approach to the software defect detection process being proposed which may be used in other similar studies. Second, the software architecture designed to demonstrate the applicability of the new approach. Third, the prototype built to evaluate the new approach. Last, the facts being observed or confirmed in the case study whose result showed that the new approach may be used to improve the performance of the software defect detection process.

7.2.1 Contributions of Approach

Observing the contradictions and drawbacks in the previous studies, this study proposed a novel approach to the software defect detection process: a self-evolving software defect detection process that has the following advantages:

1. The software defect detection process is considered as a whole and its three main activities (inspection, testing, and maintenance) are treated as being complementary to each other, instead of only studying one of them in isolation without regarding the existence of the other two or treating them as rivals by comparing their effectiveness.

2. The economics of software defect detection is taken into account by using both effectiveness and efficiency to evaluate the software defect detection process.

3. The defect detection process is conducted and controlled better by providing entrance criteria and exit criteria checking and updating.

4. An evaluation mechanism is provided by analyzing the characteristics of the defects detected during the process.

5. Continuous improvement and self-adjustment are facilitated by providing assistance to find the weak points in the current process and taking the corresponding actions.

7.2.2 Contributions of Software Architecture

This thesis presented a software architecture to implement a systematic approach to the software defect detection process by defining the necessary components, their functionalities, and the relationship between these components for the new approach. The software architecture demonstrates the applicability of the self-evolving software defect detection process approach by providing the following functionalities through its components:

1. Support the defect detection process.

2. Collect, classify, and analyze the defects.

3. Control the defect detection process.

4. Analyze and evaluate the defect detection process.

5. Continuously improve the defect detection process.

7.2.3 Contributions of Prototype and Case Study

A prototype was built and a case study was performed to evaluate the self-evolving software defect detection process approach. The preliminary results are encouraging. The prototype could be used as a starting point for implementing a self-evolving software defect detection process management system. The case study illustrates, step by step, the path that may be taken to identify the shortcoming in the software defect detection process based on the facts being observed.

7.3 Directions for Future Research

There are several directions that can be investigated in future research:

1. More case studies should be conducted to further evaluate the self-evolving software defect detection process approach.

2. An experience base should be built to assist decision-making. An experience base provides information, such as which technique helps an inspector or tester detect the most defects (i.e., maximum effectiveness) under specific conditions. For the knowledge in the experience base to be accurate and easy to retrieve, the knowledge could be stored in a highly-structured way, using the following pattern:

   Knowledge = <Solution, Issue, Context>

   - Solution: The solution to solve the issue.
   - Issue: The issue that can be solved by the solution.

- Context: The environment in which the solution is valid for the issue.

3. Mathematic models and Analytical models can be established to analyze, evaluate, and improve the software defect detection process and the self-evolving software defect detection process approach itself. These models will provide a deeper insight into the strengths and weaknesses of the current practice.

This dissertation presented preliminary research on the software defect detection process. The dissertation proposed a self-evolving software defect detection model, described the software architecture of the model, built a prototype for the model, and performed a case study for the model. Future research in this direction could help conduct, control, evaluate, and improve the software defect detection process so that it is more effective and more efficient.

References

[1] Ackerman, A. F., Buchwald, L. S., and Lewsky, F. H., Software Inspections: An Effective Verification Process. IEEE Software, 6(3), pages 31-36, 1989.

[2] Basili, V. R., Evolving and Packaging Reading Technologies. Journal of Systems and Software, 38(1), pages 3-12, July 1997.

[3] Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., and Zelkowitz, M., The Empirical Investigation of Perspective-Based Reading, Empirical Software Engineering, 1(2), pages 133-164, 1996.

[4] Basili, V. R., Selby, R. W., Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering, 13(12), pages 1278-1296, December 1987.

[5] Beizer, B., Software Testing Techniques, Second Edition, Van Nostrand Reinhold, New York, New York, 1990.

[6] Bisant, D. B., Lyle, J. R., A Two-person Inspection Method to Improve Programming Productivity. IEEE Transactions on Software Engineering, 15(10), pages 1294-1304, October 1989.

[7] Boehm, B., Basili , V. R., Software Defect Reduction Top 10 List, IEEE Computer, 34(1), pages 135-137, January 2001

[8] Businessobjects, http://www.businessobjects.com/

[9] Butcher, M., Munro, H., Kratschmer, T., Improving Software Testing via ODC: Three Case Studies - Orthogonal Defect Classification, IBM Systems Journal, March 2002.

[10] Cavano, J. P., LaMonica, F. S., Quality Assurance in Future Development Environments. IEEE Software, 4(5), pages 26-34, September 1987.

[11] Chaar, J. K., On the Evaluation of Software Inspections and Tests, In Proceedings of the 1993 International Test Conference, pages 180-189, June 1993.

[12] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., Wong, Man-Yuen, Orthogonal Defect Classification: A concept for In-Process Measurements. IEEE Transactions on Software Engineering, 18(11), pages 943-956, November 1992.

[13] Chusho, T., Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing, IEEE Transactions on Software Engineering, 13(5), pages 509-517, May 1987.

[14] Ciolkowksi, M., Differding, C., Laitenberger, O., and Munch, J., Empirical Investigation of Perspective-based Reading: A Replicated Experiment, Technical Report ISERN-97-13, 1997.

[15] Cognos, http://www.cognos.com/products/index.html.

[16] Dunsmore, A., Roper M., Wood M., Further Investigation into the Development and Evaluation of Reading Techniques for OO Code Inspection. Proceedings of the 24th International Conference on Software Engineering, pages 47-57, 2002.

[17] Dunsmore, A., Roper M., Wood M., Systematic Object-Oriented Inspection - An Empirical Study. Proceedings of the 23rd International Conference on Software Engineering, pages 135-144, 2001.

[18] Eickelmann, N. S., Ruffolo, F., Baik, J., Anant, A., An Empirical Study of Modifying the Fagan Inspection Process and the Resulting Main Effects and Interaction Effects Among Defects Found, Effort Required, Rate of Preparation and Inspection, Number of Team Members and Product 1st Pass Quality, Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, pages 58-64, 2002.

[19] Emam, K., Drouin, J., Melo, W., SPICE: The Theory and Practice of Software Process Improvement and Capability Determination, Wiley-IEEE Computer Society Press, November 1997.

[20] Ernst, D., Houdek, F., Schwinn, T., An Experimental Comparison of Static and Dynamic Defect Detection Techniques, 11th International software quality week, May 1998.

[21] Fagan, M. E., Advances in Software Inspections. IEEE Transactions on Software Engineering, 12(7), pages 744-751, 1986.

[22] Fagan, M. E., Design and Code Inspections to Reduce Errors in Program Development, IBM Systems Journal, 15(3), pages 258-287, 1976.

[23] Frankel, P. G., Weyuker, E. J., Data Flow Testing in the Presence of Unexecutable Paths, Proceedings of the Workshop on Software Testing, Banff, Canada, pages 4-13, July 1987.

[24] Fredericks, F., Basili, V., Using Defect Tracking and Analysis to Improve Software Quality, A DACS State-of-the-Art Report, University of Maryland, November 1998.

[25] Freedman, D. P., Weinberg, G. M., Handbook of Walkthrough, Inspections and Technical Reviews, Little, Brown and Company, Boston, 1982.

[26] Fusaro, P., Lanubile, F., Visaggio, G., A Replicated Experiment to Assess Requirements Inspection Techniques, Empirical Software Engineering, 2(1), pages 39-57, 1997.

[27] Garcia, R. E., Oliveira, M. C., Maldonado, J. C., Mendonc M., Visual Analysis of Data from Empirical Studies, International Workshop on Visual Languages and Computing, September 2004.

[28] Gilb, T., Graham, D., Software Inspection. Addison-Wesley, 1993.

[29] Grady, Robert B., Practical Software Metrics for Project Management and Process Improvement, Prentice Hall, 1992.

[30] Grady, Robert B., Caswell, D., L. Software Metrics: Establishing a Company-Wide Program, Prentice Hall, Englewood Cliffs, NJ, 1987.

[31] Haase, V., Messnarz, R., Koch, G., Kugler, H. J., Decrinis, P., Bootstrap: Fine-Tuning Process Assessment. IEEE Software. 11(4), pages 25-35, July 1994.

[32] Hetzel, W. H., An Experimental Analysis of Program Veification Method. PhD thesis, University of North Carolina at Chapel Hill, 1976.

[33] Humphrey, W. H., A Discipline for Software Engineering, Addison-Wesley Publishing Company, 1995.

[34] IEEE Standard 1220-1998, IEEE Standard for Application and Management of the Systems Engineering Process, Institute of Electrical and Electronics Engineers, 1998.

[35] Johnson, P., An Instrumented Approach to Improving Software Quality Through Formal Technical Review, Proceeding of 16th International Conference on Software Engineering, pages 113-122, 1994

[36] Kamsties, E., Lott, C. M., An Empirical Evaluation of Three Defect-detection Techniques. Proceedings of the Fifth European Software Engineering Conference, pages 362-383, 1995.

[37] Kan, S. H., Parrish, J., Manlove, D., In-process Metrics for Software Testing, IBM Systems Journal, 40(1), pages 220- 241, 2001.

[38] Kelly, J., Inspection and Review Glossary Part 1, Jet Propulsion Laboratory. Reprinted from SIRO Newsletter Volume 2, April 1993.

[39] Kit, E., Software Testing in the Real World, Addison-Wesley, 1995.

[40] Knight, J. C., Myers, E. A., An Improved Inspection Technique, Communications of ACM, 36(11), pages 51-61, November 1993.

[41] Kuvaja, P., Simila, J., Krzanik, L., Bicego, A., Saukkonen, S., and Koch, G., Software Process Assessment and Improvement, The Bootstrap Approach, Oxford, Blackwell, 1994.

[42] Laitenberger, O., DeBaud, J., Perspective-based Reading of Code Documents at Robert Bosch GmbH, Information and Software Technology, 39(11), pages 781-791, 1997.

[43] Laitenberger, O., El-Eman, K, Harbich, T. G., An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-Based Reading of Code Documents, IEEE Transactions on Software Engineering, 27(5), pages 387-421, 2001.

[44] Macdonald, F., Miller, J., A Comparison of Tool-based and Paper-based Software Inspection. Empirical Software Engineering 3(3), pages 233-253, 1998.

[45] Marciniak, J., Encyclopedia of software engineering, volume 1-2, John Wiley and Sons, Inc, New York, 1994.

[46] Marick, B., New Models for Test Development, Quality Week '99 - a rebuttal of the V-model.

[47] McCabe, T., A Complexity Measure, IEEE Transactions on Software Engineering, pages 308 - 320, December 1976.

[48] McCabe, T., Automating the Testing Process Through Complexity Metrics, Conference Proceedings Software Testing and Validation September 23-24, 1987, National Institute for Software Quality and Productivity, Inc., 1987, pages G-l - G-30.

[49] McCabe, T., Structured Testing, IEEE Computer Society Press, Silver Spring, Maryland, 1982.

[50] Halling, M., Biffl, S., Grechenig, T., and Kohle, M., Using Reading Techniques to Focus Inspection Performance, In Proceedings of 27th Euromicro Workshop Software Process and Product Improvement, pages 248-257, 2001.

[51] Miller, J., Wood, M., Roper, M., Further Experiences with Scenarios and Checklists, Journal of Empirical Software Engineering, 3(3), pages 37-64, 1998.

[52] Musa, J. D., Iannino, A., Kazuhira, O., Software Reliability: Measurement, Prediction, Application. McGraw-Hill, 1990.

[53] Myers, G. J., A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections, Communications of ACM, 21(9), pages 760-768, September 1978.

[54] Myers, G. J. The Art of Software Testing, John Wiley and Sons, New York, New York, 1979.

[55] Paulk, M., Curtis, B., Chrissis, M., The Capability Maturity Model: Guideline for Improving the Software Process, Addison-Welsley, 1995.

[56] Piwowarski, P., Ohba, M., Caruso, J., Coverage Measurement Experience During Function Test. In Proceedings of the 15th International Conference on Software Engineering, 1993.

[57] Porter, A., Votta, L. G., An Experiment to Assess Different Detection Methods for Software Requirements Inspection, Proceedings of the 16th International Conference on Software Engineering, pages 103-112, Sorrento, Italy, 1994.

[58] Porter, A., Votta, L. G., Basili V R. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, IEEE Transactions on Software Engineering, 21(6), Pagers 563-575, June 1995.

[59] Porter, A., Votta, L. G., Comparing Detection Methods for Software Requirement Inspections: A Replication Using Professional Subjects, Empirical Software Engineering Journal 21(6), pages 355-79, 1998.

[60] Porter, A., Votta, L. G., Basili V. R., Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, IEEE Transactions on Software Engineering, 21(6), pages 563-575, 1995.

[61] Sabaliauskaite, G., Matsukawa, F., Kusomoto S., Inoue K., An Experimental Comparison of Checklist-Based Reading and Perspective-Based Reading for UML Design Document Inspection, In Proceedings of the International Symposium on Empirical Software Engineering, pages 148-157, Nara, Japan, 2002.

[62] Sandahl, K., Blomkvist, O., Karlsson, J., Krysander, C., Lindvall M., Ohlsson, N., An Extended Replication of an Experiment for Assessing Methods for Software Requirements Inspections, Empirical Software Engineering 3(4), pages 327-354, 1998.

[63] Schneider, G. M., Martin, J., and Tsai, W., An Experimental Study of Fault Detection in User Requirements Documents, ACM Transactions on Software Engineering, Methodol. 1(2), pages 188-204, April 1992.

[64] Selby, R.W., Combining Software Testing Strategies: An Empirical Evaluation. IEEE Workshops on Software Testing, 36(11), pages 82-90, July 1986.

[65] Software Formal Inspections Guidebook NASA Office of Safety and Mission Guidance. NASA-GB-A302, 1993.

[66] Simila, S., Kuvaja, P., Krzanik, L., BOOTSTRAP: A Software Process Assessment and Improvement Methodology, Proceedings of the First Asia-Pacific Software Engineering Conference, pages 183-196, 1994.

[67] Sommerville, I., Software Engineering, 6th edition. Chapter 19 Software inspections, 2000.

[68] Tervonen, I., Support for Quality-Based Design and Inspection. IEEE Software, 13(1), pages 44-54, 1996.

[69] Thelin T, Runeson P., Wohlin C., Prioritized Use Cases as a Vehicle for Software Inspections. IEEE Software, 20(4), pages 30-33, July/August 2003.

[70] Turing, A., Checking a Large Routine, Report of a Conference on High Speed Automatic Calculating-Machines, pages 67-69, January 1950.